Spring 6-2-2017

An Efficient Pipeline for Assaying Whole-Genome Plastid Variation for Population Genetics and Phylogeography

Brendan F. Kohrn
Portland State University

An Efficient Pipeline for Assaying Whole-Genome Plastid Variation for Population Genetics and Phylogeography

by

Brendan F. Kohrn

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Biology

Thesis Committee:
Mitch Cruzan, Chair
Sarah Eppley
Rahul Raghavan

Portland State University
2017

Abstract

Tracking seed dispersal using traditional, direct measurement approaches is difficult and generally underestimates dispersal distances. Variation in chloroplast haplotypes (cpDNA) offers a way to trace past seed dispersal and to make inferences about factors contributing to present patterns of dispersal. Although cpDNA generally has low levels of intraspecific variation, this can be overcome by assaying the whole chloroplast genome. Whole-genome sequencing is more expensive, but resources can be conserved by pooling samples. Unfortunately, haplotype associations among SNPs are lost in pooled samples and treating SNP frequencies as independent estimates of variation provides biased estimates of genetic distance. I have developed an application, CallHap, that uses a least-squares algorithm to evaluate the fit between observed and predicted SNP frequencies from pooled samples based on network topology, thus enabling pooling for chloroplast sequencing for large-scale studies of chloroplast genomic variation. This method was tested using artificially-constructed test networks and pools, and pooled samples of *Lasthenia californica* (California goldfields) from Whetstone Prairie, in Southern Oregon, USA. In test networks, CallHap reliably recovered network topologies and haplotype frequencies. Overall, the CallHap pipeline allows for the efficient use of resources for estimation of genetic distance for studies using non-recombining, whole-genome haplotypes, such as intra-specific variation in chloroplast, mitochondrial, bacterial, or viral DNA.

Dedication

This thesis is dedicated to my grandfather, Dr. Lawrence Loeb, and to my parents, Corinne and David Kohrn, for helping me through the long educational trajectory that led me to where I am now, and for their continued support as I move into the future.

Acknowledgments

I would like to acknowledge Mitch Cruzan, for helping talk me through different ideas for the pipeline and keeping me focused through the long hours of debugging, Pam Thompson, Monica Grasty, Tina Arredondo, Jaime Schwoch, and Elizabeth Hendrickson, for helping me work through issues with the program and suggesting features, Jessica Persinger for her help with *de novo* assemblies, my parents and grandparents, for helping support me through my thesis research, and PSU Academic and Research Computing, for making sure the servers kept running while I was developing this pipeline.

This thesis was completed with funding from a NSF Macrosystems Biology grant (award number 1340746)

Table of Contents

List of Tables

List of Figures

Introduction

Gene flow includes a number of processes that cause changes in allele frequencies, including movement of gametes (gametophytes) or individuals across the physical landscape (Slatkin, 1987). In the case of plants, movement by gametes is restricted to dispersal of pollen (the male gametophyte) from the location of the paternal individual to the maternal individual, and dispersal of individuals is reduced to movement of seeds. All other life stages of plants are sessile, or have limited mobility through vegetative growth. The distribution of genetic variation within and among plant populations from gene flow can thus be reduced to seed and pollen dispersal. Of these, only seed dispersal has the potential to establish new populations through colonization of vacant sites (Howe and Smallwood, 1982; Nathan and Muller-Landau, 2000).

The movement of seeds to a new site from the location of the maternal parent can occur through a variety of vectors. Some plants have seeds designed to float on the wind (anemochory), while others have seeds which attach themselves to the outside of an animal (ectozoochory), have fleshy fruits meant to be eaten by animals (endozoochory), or just fall off the maternal plant under the influence of gravity (barochory) (Howe and Smallwood, 1982). Traditionally, seed dispersal has been measured by direct observation using a variety of seed trap designs (Gorchov et al., 1993; Kollmann and Goetze, 1998; Godoy and Jordano, 2001), by testing for the presence of seeds in the feces of local herbivorous species (Mouissie et al., 2005), or by observing the movement patterns of dispersal vectors (Kays et al., 2011). Unfortunately, these approaches can be difficult to

implement and tend to overestimate short-distance seed dispersal while missing long-distance dispersal events (Willson, 1993).

Long-distance seed dispersal is particularly difficult to measure through direct observation approaches, and may be disproportionately important for gene flow and establishing new populations (Cain et al., 2000; Trakhtenbrot et al., 2005). Although measuring seed dispersal is often difficult, genetic markers can be used to track patterns and intensity of historical dispersal. Nuclear genetic markers, including most single nucleotide polymorphisms (SNPs) and microsatellites, offer one potential solution, but variation in these markers within and among populations is affected by both pollen and seed dispersal. In contrast, chloroplast DNA (cpDNA) is inherited maternally in most angiosperms (Corriveau and Coleman, 1988), which means variation in these markers is only affected only by the process of seed dispersal.

Using cpDNA variation (SNPs) to measure genetic distance presents a few challenges. First, chloroplast genomes are non-recombining and effectively haploid (Palmer, 1987), so SNPs common to the same haplotype are inherited together. This feature allows for the reconstruction of network phylogenies that illustrate the relationships among haplotypes, but means that, no matter how many cpDNA SNPs are found, the whole chloroplast can only be treated as a single locus. I found that treating cpDNA SNPs as independent markers will tend to underestimate levels of differentiation and genetic distances among populations, especially when haplotypes share SNPs (Fig. 1). In the past, cpDNA markers have not been considered very useful due to the slow evolutionary rate of chloroplast genomes, which results in low intra-specific variation

(Palmer, 1987). Modern sequencing methods alleviate this problem by allowing the detection of larger numbers of SNPs across the entire chloroplast genome (Stull et al., 2013). Combinations of SNPs represent chloroplast haplotypes, and are a valuable tool for examining genetic diversity and seed dispersal in angiosperms.

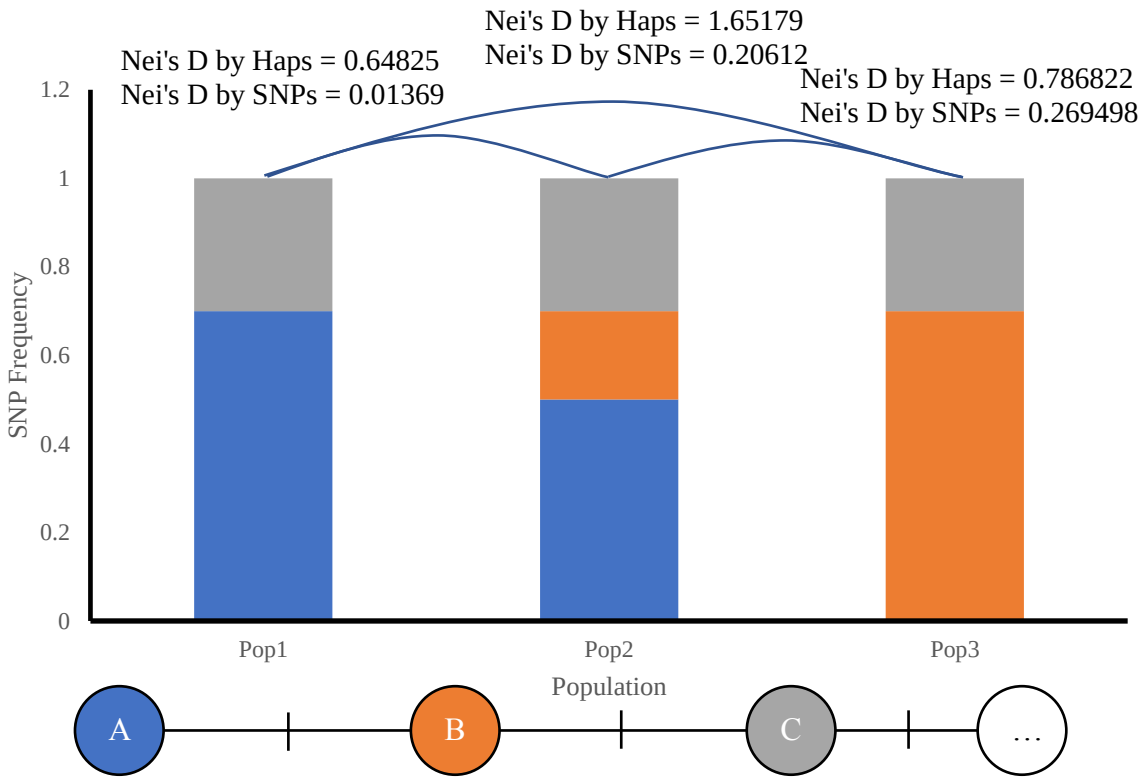


Fig. 1. SNP frequency contribution from multiple haplotypes where a SNP is shared between haplotypes. In this case, each population contains the same three haplotypes, with one being found at a constant frequency in all three populations, while the other two, which share a SNP, are found at varying frequencies in the three populations, such that the overall frequency of that SNP is constant. A network phylogeny showing the three haplotypes and their relatedness to each other is shown below the figure.

When using chloroplast haplotypes for population genetics and phylogeographic studies, cpDNA from many individuals must be sequenced to generate adequate sample sizes for the estimation of genetic parameters. Although sequencing costs have decreased in recent years, sequencing enough samples for a large-scale population genetics study

still requires significant resource investment (Sboner et al., 2011). Pooling multiple individuals for sequencing has become common as a solution to this problem (Sham et al., 2002; Schlötterer et al., 2014). Unfortunately, pooling cpDNA samples results in the loss of information about the SNP associations that represent each haplotype because DNA sequencing only recovers SNP frequencies (Fig. 1). While there are a number of haplotype reconstruction programs available, these are either aimed exclusively at diploid genomes or at resolving (nuclear) haplotypes over smaller genomic regions (i.e. phasing; Pe'er and Beckmann, 2003; Kirkpatrick et al., 2007; Gasbarra et al., 2011; Kofler et al., 2011). These methods assume some level of recombination, and ultimately are not appropriate for the recovery of haplotypes from the non-recombining chloroplast genome. To solve this problem, I have developed a new bioinformatics pipeline aimed at reducing the cost of population-level surveys of chloroplast diversity by reconstructing chloroplast haplotypes from pooled samples using a sample of sequenced individual chloroplast haplotypes.

Here, I describe sampling and bioinformatics protocols for the examination of haplotype-based population genetics, including the variant filtering and haplotype recovery programs of CallHap. I then test the CallHap haplotype recovery program using a series of artificial networks and pools. Finally, I provide an example of CallHap processing using a set of *Lasthenia californica* Lindley (Asteraceae) samples collected from Whetstone Prairie, near Medford, OR.

Bioinformatics Pipeline

Sample collection and sequencing

The CallHap pipeline (Appendix A) begins after sampling tissue from some number of individuals (e.g., 20) from each of several populations and extracting DNA from each sampled individual. Equimolar amounts of DNA from each individual are used to make pooled sequencing libraries (PLs). A representative subset of individuals sampled across populations is used to make a collection of single sample libraries (SSLs), which are used to construct a skeleton network phylogeny. Both single and pooled libraries are filtered to concentrate cpDNA using a RNA capture array (Appendix B; Stull et al., 2013), and sequenced using next-generation platforms.

Sequence data processing

Sequences are processed using cutadapt v1.9.1 (Martin, 2011) for adapter trimming and sickle v1.33 (Joshi and Fass, 2011) for quality trimming. BWA v0.7.5a (Li and Durbin, 2009) is used to align trimmed sequences to a single reference genome. Sequences were sorted using Samtools v0.1.17, read groups were added using Picard Tools v.1.141 (http://broadinstitute.github.io/picard), and indel realignment was carried out using Genome Analysis Toolkit v3.5 (GATK; McKenna et al., 2010). Initial variant calls are made using FreeBayes v1.0.2 (Garrison and Marth, 2012) or other similar programs.

Variant Filtering

Variant filtering is carried out using the first of the two major CallHap programs, CallHap_VCF_Filt.py (Program flowchart in Appendix C). This script filters raw variants to ensure that they can be used by the main haplotype caller by removing (a) non-SNP variants, due to the difficulty in calling insertion or deletion type variants (indels) as being in one of two states, (b) variants with low depth or quality, (c) variants that do not have a defined identity across all SSLs and pooled libraries, since the haplotype caller application cannot handle missing values in the matrix of haplotype identities, (e) SNPs in close proximity to indels, due to difficulties in creating correct alignments in these regions. Filters that have a limit (depth filter, indel proximity, and quality filters) are can be modified to meet the demands of a particular study. The variant filter outputs a file containing genotype data for SSLs, a separate file containing SNP frequency data for PLs, and a NEXUS file for network phylogeny creation.

Recovery of haplotypes from pooled samples

The CallHap Haplotype Caller (CallHap_HapCallr.py, Appendix D) works by iterating through the available SNPs in a pseudo-random order, with SNPs present in SSL (known) haplotypes being processed first. Processing a large number of these random orders increases certainty in haplotype calls. Within each order, an initial estimate of haplotype frequencies is generated using a least squared algorithm (Appendix E) to solve the equation $A\,x = b$, where A is the binary matrix of SNP identities in various haplotypes, x is the unknown vector of haplotype frequencies, and b is the observed vector of SNP frequencies. An overall average Residuals Sums of Squares (RSS) value

is computed by averaging RSS values based on each PL. In addition, the total RSS value for each SNP is computed.

Next, the algorithm creates new haplotypes based on each SNP for which there exists a non-zero residual in the initial solution (See Appendix F). If the current SNP is present in the known haplotypes, new haplotype creation only considers creating new haplotypes based on the haplotypes at either end of the network phylogeny branch along which this SNP occurs. Otherwise, the algorithm considers every possible new haplotype (Fig. 2). Average RSS values are computed for each possible haplotype attachment point, and the proposed new haplotype matrix and average RSS values are saved for later filtering. This procedure is repeated for each possible solution for all SNPs. Once all possible solutions have been processed for each SNP, the haplotypes matrices are filtered to only keep those that produced the lowest average RSS value (Fig. 2).

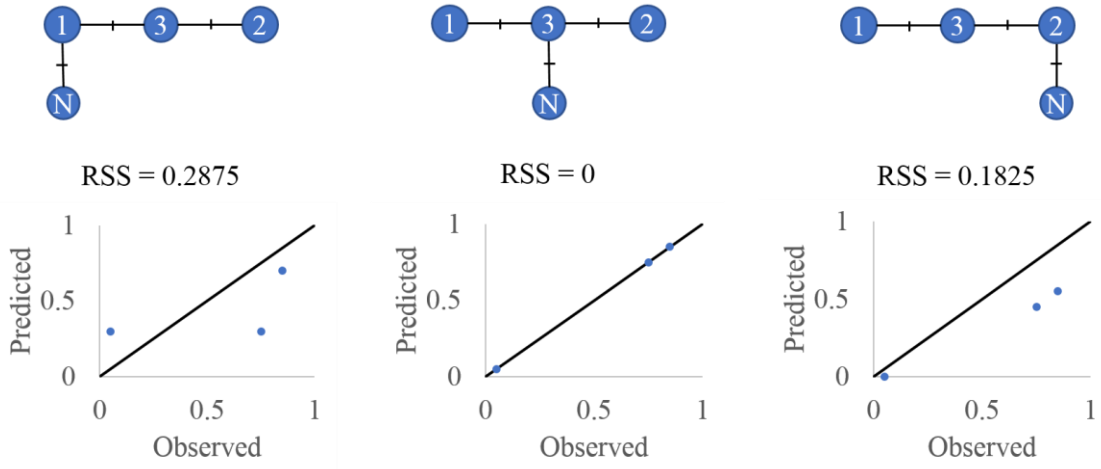


Fig. 2. Haplotype creation and selection of best position in a simple haplotype system. N, in each case, represents the position of the newly-created haplotype. Graphs show predicted vs. observed SNP frequencies.

Once all SNPs have been processed, the haplotypes matrices are filtered to remove unused haplotypes. Haplotypes matrices are then filtered to keep only those with the lowest Akaike information criterion (AIC; Li et al., 2002). The columns of these matrices (the haplotypes) are taken as binary numbers with 1 representing the reference and 0 the alternate allele, converted into decimal numbers representing the haplotypes, and saved along with the average RSS values produced by the matrices.

After completing all pseudo-random orders, output files are generated showing the raw haplotypes produced in each proposed solution, the percentage of random orderings for which a particular haplotype was produced, the number of times each unique topology was generated and the average RSS value for each, haplotype frequencies in each pool and the RSS value for that pool, VCF files showing predicted SNP frequencies in each pool and RSS for each SNP, a CSV file comparing observed vs. predicted SNP frequencies, and a NEXUS file for network phylogeny creation using PopART (http://popart.otago.ac.nz) or similar. Optionally, a genpop file that can be imported into adegenet (Jombart, 2008) and a STRUCTURE-formatted file (Pritchard et al., 2000; Raj et al., 2014) can also be generated. Haplotype frequencies are presented as number of individuals with that haplotype, and haplotypes are presented as multiple alleles at a single locus (the chloroplast).

After CallHap generates outputs users can examine the resulting topologies and select a final topology based on (1) the average RSS value of the solution, (2) the frequency with which a given topology occurred, and (3) based on the commonality of

the root haplotype for any mobile new haplotypes not resolved by the first two criteria (Templeton et al., 1992).

Artificial networks

Test network phylogenies were created to represent different types of network topologies (Fig. 3). Seven artificial pools containing twenty individuals each were created based on each network, with each pool containing three random haplotypes at frequencies approximating the Poisson distribution. Each set of artificial pools was run through the haplotype caller, using 100 random orders, and 2 iterations per order, with different combinations of "known" haplotypes to see if (a) the correct network topology was recovered by the best solution, and (b), if the correct haplotype frequencies were recovered by the best solution.

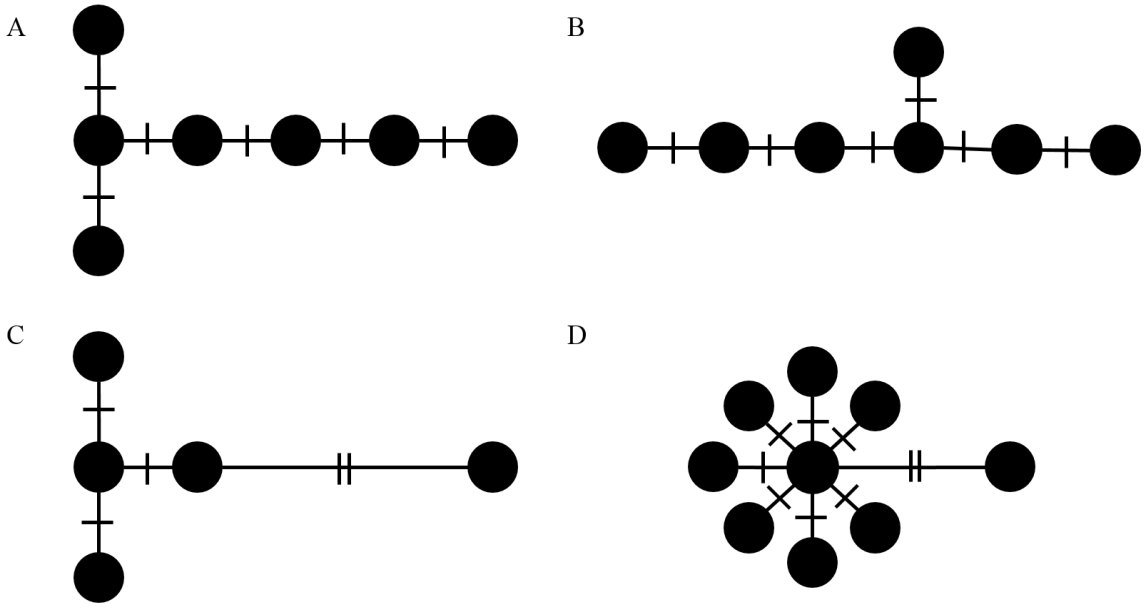


Fig. 3: Test Network Phylogenies. These phylogenies were designed to test the ability of CallHap to recover different topological patterns when starting with different haplotypes; (A) a long branch with every haplotype defined (B) two long branches with all haplotypes defined (C) a long branch with some haplotypes defined, and (D) a cluster with one haplotype further out.

Testing with *Lasthenia californica*

Leaf tissue was collected from 400 individuals across 20 populations of *Lasthenia californica* located within a 16-hectare area of Whetstone Prairie, near Medford, OR, USA (P. Thompson et. al, unpublished. data). Tissues were dried using silica beads as a desiccant, and DNA was extracted using a Qiagen Plant DNeasy 96 kit (Qiagen, Germantown, MD USA). After DNA extraction, DNA concentration was quantified on a Qubit 3.0 fluorimeter (Thermo Fisher Scientific, Waltham, MA USA), and pooled by population in an equimolar fashion. Library preparation was carried out using a NEBNext Ultra DNA Library Prep Kit (E7370) with NEBNext Multiplex Oligos (E7600; New England Biolabs, Ipswich, MA, USA). Single sample libraries were constructed for at least one individual from each population.

SSLs and PLs were captured using a MYbaits-3 custom cpDNA capture array from MYcroarray (MYcroarray, Ann Arbor, MI, USA; Appendix B). DNA was sequenced on an Illumina HiSeq 2500 Sequencer (Illumina, San Diego, California, USA) using 5 lanes, with 100bp paired-end reads generated for all but six samples, which had 100bp single end reads (Massively Parallel Sequencing Shared Resource Facility, Oregon Health and Science University). The contents of each lane are summarized in Table 1. Sequence alignment was performed both to the published *Lasthenia burkei* chloroplast genome (Walker et al., 2014) and to an in-house partial *de novo* reference for *L. californica* (KY965816). SNP calling and variant filtering were performed on both alignments using the pipeline described above with a minimum read depth of 600 and a minimum variant quality of 20. Haplotype calling was performed using information from

de novo alignments. For the full dataset, haplotype calling was run a second time with any new haplotypes that were consistently added placed in the input haplotypes to help resolve mobile haplotypes.

Table 1: Summary of sequencing lane contents, showing number of *Lasthenia californica* SSLs and PLs used in analysis on each lane, number of other libraries on each lane, percentage *L. californica* returns from each lane, and type (single end or paired end) of each run

Lane	*L. californica* # SSLs[1]	*L. californica* # PLs[1]	Other Libraries[2]	% Returns *L. californica*[1]	Run Type
1	5	0	1	99.02%	se
2	13	4	7	61.39%	pe
3	20	0	28	17.53%	pe
4	7	16	31	12.42%	pe
5	2	0	52	2.14%	pe

[1] Number only reflects libraries used in analysis

[2] These libraries were made using species other than *L. californica*, or were *L. californica* libraries unused in this analysis.

Results

Test Networks

Correct haplotype networks were recovered as single lowest RSS value solutions in all starting conditions for three out of four test networks. For the fourth, the correct haplotype network was recovered as the more common of two possible solutions with the lowest RSS value (Fig. 4).

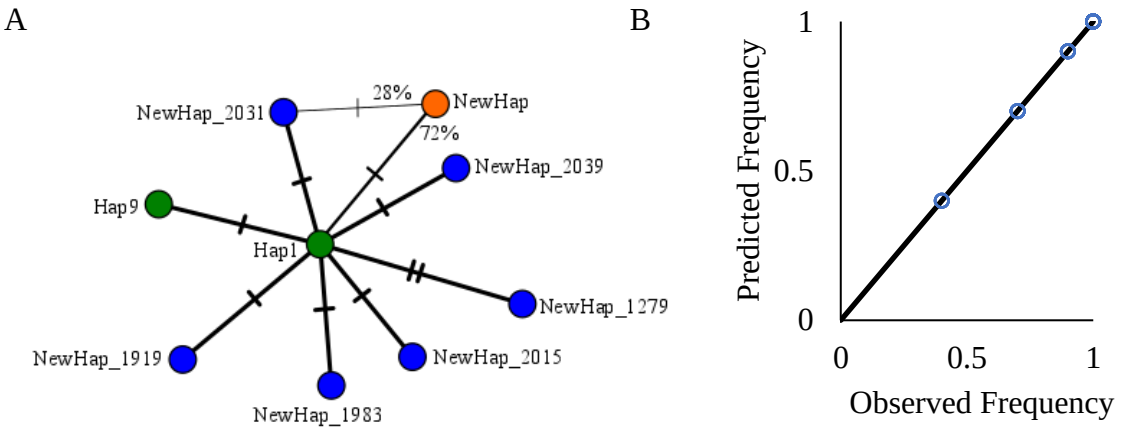


Fig. 4. Resulting phylogeny from one starting condition from Test Network D. (A) Green haplotypes were known at the beginning, blue haplotypes were present in all solutions at the lowest RSS value, and orange haplotypes had ambiguous positions between different solutions. Branch thicknesses are scaled by how many times a solution with the branch occurred, and percentages give exact percent of time a branch occurred. Hash marks indicate number of SNPs along a branch. (B) Regression plot for these solutions.

Lasthenia californica testing

Sequencing – Sixty-seven libraries (47 SSLs and 20 PLs) were sequenced, producing a total of 753,355,673 raw reads. Of these, 88% of raw reads mapped to the *L. burkei* genome, while 85% of raw reads mapped to the *L. californica de novo* genome (Table 2).

Table 2: Summary of sequencing data for Whetstone Prairie *L. californica* libraries.

Location #	**Individual #**	**SSL/PL**	**Type**	**Raw Reads**	**% mapped (*L. burkei*)**	**% mapped (*de novo*)**
1	5	SSL	PE	2,204,954	86.26%	89.11%
1	-	PL	PE	2,838,092	88.45%	80.80%
2	20	SSL	PE	3,542,874	87.12%	88.29%
2	-	PL	PE	1,894,746	87.77%	81.50%
3	17	SSL	PE	11,294,164	86.17%	86.89%
3	-	PL	PE	2,046,704	86.31%	80.49%
4	5	SSL	SE	48,910,678	90.81%	88.66%
4	-	PL	PE	2,101,514	88.19%	92.17%
5	8	SSL	SE	34,911,454	89.85%	86.48%
5	-	PL	PE	1,517,502	85.58%	91.55%
6	2	SSL	SE	44,544,597	90.29%	88.93%
6	-	PL	PE	4,198,354	88.44%	91.93%
7	5	SSL	PE	7,489,900	86.49%	87.94%
7	-	PL	PE	3,589,012	87.38%	80.09%
8	2	SSL	PE	2,722,540	89.00%	79.70%
8	8	SSL	SE	86,867,128	91.01%	89.43%
8	9	SSL	PE	2,348,132	87.79%	92.58%
8	12	SSL	PE	1,636,222	88.94%	88.24%
8	17	SSL	PE	1,748,016	89.70%	89.38%
8	18	SSL	PE	1,398,104	89.73%	90.14%
8	-	PL	PE	10,422,822	86.47%	90.17%
9	6	SSL	SE	30,468,788	90.13%	81.60%
9	-	PL	PE	7,839,014	87.23%	91.86%
10	8	SSL	PE	49,460,308	87.26%	86.26%
10	-	PL	PE	1,836,746	85.75%	80.79%
11	19	SSL	PE	6,102,332	89.95%	87.85%
11	-	PL	PE	2,141,130	87.33%	90.48%
12	4	SSL	PE	2,684,862	90.09%	88.83%
12	-	PL	PE	1,819,176	88.29%	90.58%
13	19	SSL	PE	20,000,548	83.59%	87.93%

Location #	Individual #	SSL/PL	Type	Raw Reads	% mapped (*L. burkei*)	% mapped (*de novo*)
13	-	PL	PE	1,426,644	87.34%	73.74%
14	16	SSL	PE	30,605,932	84.92%	90.35%
14	-	PL	PE	5,877,608	89.81%	77.81%
15	1	SSL	PE	3,262,234	89.21%	89.69%
15	2	SSL	PE	2,704,746	85.61%	86.14%
15	4	SSL	PE	6,493,262	85.65%	86.12%
15	5	SSL	PE	4,483,434	89.56%	90.09%
15	6	SSL	PE	7,358,804	89.29%	89.82%
15	6	SSL	PE	5,371,566	88.69%	89.17%
15	7	SSL	PE	5,588,912	88.00%	88.35%
15	8	SSL	PE	1,331,344	88.81%	89.31%
15	9	SSL	PE	2,146,432	87.41%	87.94%
15	9	SSL	PE	6,855,194	88.15%	88.57%
15	10	SSL	PE	15,648,756	89.90%	81.29%
15	10	SSL	PE	8,371,854	87.96%	88.32%
15	11	SSL	PE	3,635,032	89.75%	90.17%
15	12	SSL	PE	3,030,866	89.05%	89.57%
15	13	SSL	PE	2,932,830	89.33%	89.79%
15	15	SSL	PE	2,565,608	90.12%	90.56%
15	16	SSL	PE	7,809,292	88.34%	88.99%
15	16	SSL	PE	6,677,584	88.97%	89.67%
15	17	SSL	PE	2,207,558	88.91%	89.38%
15	18	SSL	PE	7,953,306	89.29%	90.09%
15	18	SSL	PE	1,585,198	89.78%	90.29%
15	19	SSL	PE	3,384,392	90.23%	90.68%
15	20	SSL	PE	6,599,688	90.22%	90.67%
15	-	PL	PE	6,700,776	87.11%	81.86%
16	19	SSL	PE	15,407,184	86.15%	80.04%
16	-	PL	PE	1,545,786	86.87%	87.50%
17	12	SSL	PE	29,139,552	84.92%	75.71%
17	-	PL	PE	6,241,342	88.75%	89.31%
18	2	SSL	PE	40,660,014	86.32%	77.87%
18	-	PL	PE	5,805,872	89.23%	89.82%
19	3	SSL	PE	33,083,718	86.01%	77.69%
19	-	PL	PE	6,008,330	88.95%	89.47%
20	12	SSL	PE	30,786,938	86.17%	78.63%
20	-	PL	PE	13,827,464	86.09%	77.99%

De novo vs. non-de novo alignment – For *L. californica* aligned to *L. burkei*, initial variant calling revealed 3154 variants, many of which represented differences between *L. californica* and *L. burkei*. After variant filtering, 34 SNPs in 16 unique haplotypes were identified across sampled populations of *L. californica*. Comparatively, for *L. californica* alignment to in house *de novo*, 978 initial variants were recovered, which simplified to 39 SNPs in 19 unique haplotypes after filtering. Of these, 26 appeared to be identical to SNPs from the *L. burkei* alignment

Initial haplotype calling for the complete *L. californica* data recovered two solutions at a minimum RSS value of 0.003002, with seven new haplotypes common to all the top three solutions and three unfixed haplotypes. Rerunning with the common haplotypes added to the SSL haplotypes returned a solution with a RSS value of 0.003002, a solution with a RSS value of 0.003077, and a solution with a RSS value of 0.003165; these topologies are summarized in Fig. 5, and RSS values are summarized in Table 3. Although the best RSS value solution wasn't the most common solution, the difference in the RSS values was small enough that the solutions are essentially equivalent. Additionally, there were only minor differences in haplotype frequency between the best RSS value solution and the second best RSS value solution. Since the RSS values for the best two solutions were so similar, the more common topology was selected as the best topology.

Average phylogenetic distance was calculated between haplotypes in each pair of populations (Between) or between haplotypes within a single population (Within) using the formula:

$$\text{Average \% difference}_{a,b} = \sum_i \sum_j d_{i,j} * p_{i,a} * p_{j,b}$$

Where i and j are haplotypes, a and b are populations, $p_{i,a}$ is the frequency of haplotype i in population a, and $d_{i,j}$ is the number of SNPs different between haplotypes i and j. This showed that haplotypes within a population were more similar to each other than haplotypes in different populations (2-sample t-test, df=25, t = 6.49, $p << 0.01$; Fig. 6).

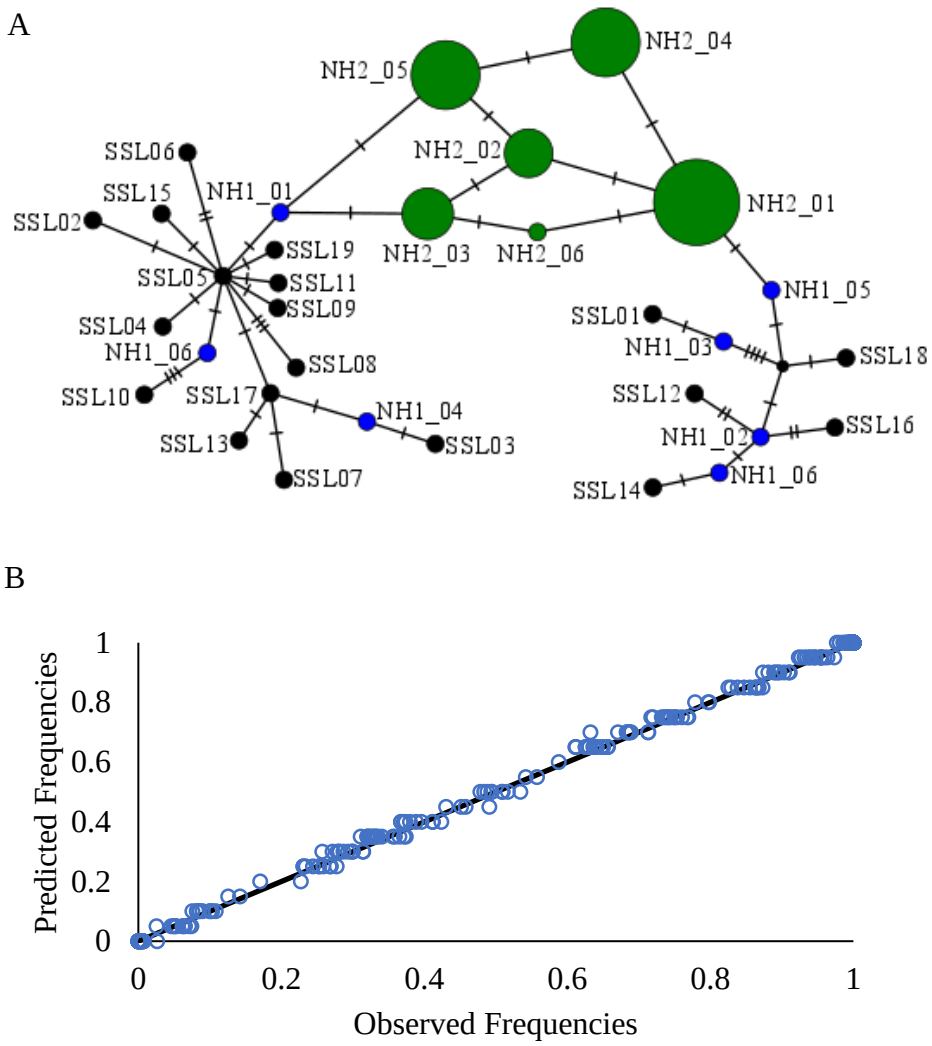


Fig. 5. Haplotypes solution for *L. californica de novo* alignment. (A) Consolidated network phylogeny for CallHap solutions with the lowest RSS value (0.003002). Black indicates starting haplotypes, blue indicates new haplotypes fixed in the best solutions from the initial haplotype calling run, and green indicates new haplotypes found in the second haplotype calling run. For the second run, node size is scaled to indicate the number of output solutions a new haplotype occurred in. Hash marks indicate number of SNPs along a branch. (B) Regression plot for lowest-RSS value CallHap solutions.

Table 3: RSS values and residual statistics (A) on a per-SNP basis, and (B) on a per-population basis. A squared residual value of 0.0025 is equivalent to one individual's worth of error.

A

SNP #	RSS	Average squared residual	Standard Deviation of squared residuals
0	0.000176	0.000009	0.000032
1	0.002585	0.000129	0.000429
2	0.000040	0.000002	0.000006
3	0.000128	0.000006	0.000026
4	0.000071	0.000004	0.000012
5	0.000459	0.000023	0.000058
6	0.001599	0.000080	0.000154
7	0.004566	0.000228	0.000262
8	0.001141	0.000057	0.000142
9	0.006557	0.000328	0.000486
10	0.004619	0.000231	0.000390
11	0.000200	0.000010	0.000043
12	0.000147	0.000007	0.000032
13	0.002009	0.000100	0.000294
14	0.001082	0.000054	0.000141
15	0.000552	0.000028	0.000107
16	0.001887	0.000094	0.000249
17	0.002112	0.000106	0.000239
18	0.000791	0.000040	0.000099
19	0.002005	0.000100	0.000198
20	0.000606	0.000030	0.000134
21	0.000714	0.000036	0.000119
22	0.003955	0.000198	0.000366
23	0.000143	0.000007	0.000028
24	0.000416	0.000021	0.000090
25	0.004510	0.000226	0.000283
26	0.000026	0.000001	0.000002
27	0.010800	0.000540	0.001008
28	0.000448	0.000022	0.000085
29	0.000131	0.000007	0.000008
30	0.001441	0.000072	0.000318
31	0.000947	0.000047	0.000145
32	0.000180	0.000009	0.000036
33	0.000173	0.000009	0.000024
34	0.000744	0.000037	0.000156
35	0.000354	0.000018	0.000011
36	0.000064	0.000003	0.000008
37	0.003147	0.000157	0.000276
38	0.000020	0.000001	0.000001

B

Population	RSS Value
1	0.004688
2	0.000322
3	0.005565
4	0.001729
5	0.005304
6	0.002121
7	0.000042
8	0.003693
9	0.000446
10	0.005215
11	0.004026
12	0.006501
13	0.003062
14	0.004435
15	0.000325
16	0.002084
17	0.000382
18	0.006086
19	0.001960
20	0.003560

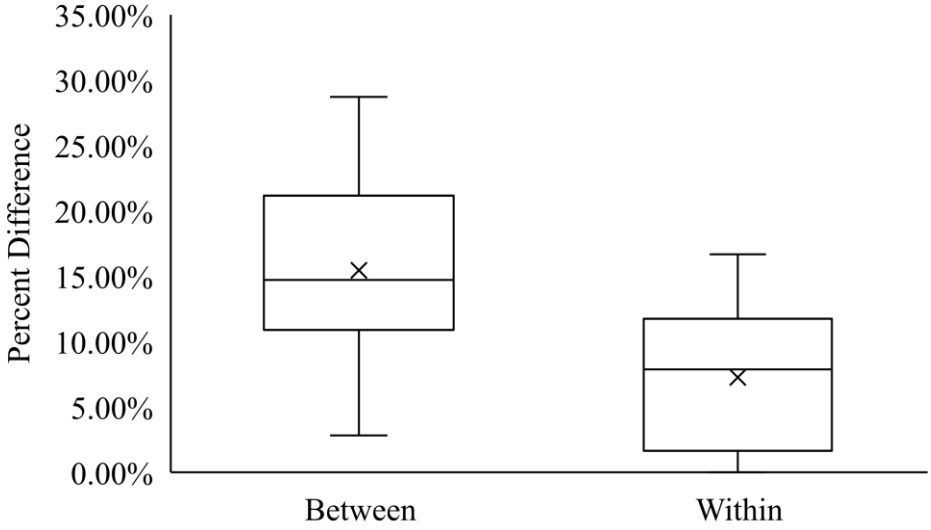


Fig. 6. Average % difference between haplotypes within populations vs. between populations.

Discussion

I have developed a pipeline, CallHap, for efficient examination of cpDNA variation, and tested it using a variety of test networks and a real data set of *Lasthenia californica* samples from Whetstone Prairie. Here, I present (A): an examination of test network results, (B): considerations for the design of experiments using CallHap, and (C): appropriate protocols for analysis of CallHap outputs. In addition, I provide an explanation for the magnitude of RSS values calculated by CallHap.

Test Networks Results

Examination of the test network pools shows consistent recovery of haplotype networks from a starting point of two or more haplotypes in the absence of any sequencing error. The presence of two possible solutions in the fourth test network reveals one potential problem that could arise during haplotype construction; if the frequencies for a new haplotype (based on SNPs not present in any of the SSLs) are less than the frequencies for multiple other haplotypes across all PLs, it is possible the new haplotype may be placed ambiguously between multiple locations on the network. When the false haplotype position was not one of the known haplotypes, the correct solution was the more common solution. One solution to this issue would be to add new haplotypes that were found consistently between the solutions with the best RSS values to the starting haplotypes array and rerunning the program. By using the expanded array of haplotypes as a starting point, differences between solutions with the same RSS value may be resolved. Another method involves taking the source DNA samples and creating

extra PLs by reshuffling the samples in ways that don't reflect the geographic areas the samples were collected in (discussed in more detail later).

Testing also revealed that, with minimal sampling of SSLs, convergence to a best solution was proportional to the centrality of the starting haplotype. As an example, for one of the test pools, all 100 orders converged to the lowest RSS value when the starting haplotype was the most central haplotype, as opposed to 13/100 and 3/100 for starting haplotypes one and two SNPs different from the most central haplotype, respectively. Also, the presence of long branches in the correct topology reduced the frequency with which that topology came up. In cases where CallHap is finding a large number of topologies, rerunning CallHap with a larger number of random orderings, potentially in combination with augmenting the known haplotypes with any haplotypes found universally, may help. In addition, starting with more than one SSL per population will increase the likelihood that the most central haplotype will be included in the SSL haplotypes.

It is apparent from examining the inferred haplotype frequencies for *L. californica* that RSS values for individual populations differ substantially. There can be many reasons for this; in some cases, high RSS values may be due to a low-quality SNP that was not filtered out correctly. For this reason, even after automated SNP filtering, any remaining SNPs should be visualized using IGV (Thorvaldsdóttir et al., 2013) or other similar programs to ensure quality. Potential issues include SNPs that occur at approximately the same frequency across populations while the other SNPs in the pool

change frequencies (especially if the major SNP present in the pool changes frequency). In these cases, the inconsistent SNPs are most likely artificial and should be removed.

Another potential cause of high RSS values is a large number of SNPs present at a frequency of more than 1/*n*, where *n* is the number of individuals in a pool. For example, if a pool contains only three SNPs at such frequencies, an RSS of 0.05 could indicate a problem; for a 20-individual pool, a RSS value of 0.0025 is equivalent to one individual-worth of error, so a RSS value of 0.05 under these conditions would indicate an average error of +/- 6.7 individuals for each haplotype present in that pool. If the same RSS value were to occur in a pool where 20 SNPs were present at these frequencies, it would be less of a problem because it would indicate an average error in haplotype frequencies of +/- 1 individual. In the *L. californica* data, the average RSS value was 0.003077, and was less than one individual's worth of error per haplotype present in all of the pools.

Experimental design considerations for CallHap Analyses

When designing an experiment to feed into the CallHap pipeline, consideration must be given to (1): the spatial scale of sampling, (2): the number of populations sampled, and (3): the size of pooled libraries. In addition, the choice of reference genome for sequence alignment and variant discovery, and the minimum read depth used, is important and needs to be contemplated.

Spatial Scale of Sampling – Experimental designs which produce data for the CallHap pipeline will differ primarily on the geographic scale of sampling. For this purpose, small-scale sampling indicates that populations are sampled at distances smaller than the hypothesized average dispersal distance of the target species, and large-scale

sampling indicates that populations are sampled at distances greater than the hypothesized average dispersal distance of the target species. At small scales, dispersal is great enough that each haplotype may be found in any location so populations are differentiated primarily by differences in the frequencies of shared haplotypes, meaning that experiments should be designed with one SSL and one PL per population. In this type of experiment there is a lowered likelihood of difficulties in recovering the correct network topology and frequencies.

At large scales, populations in close proximity to each other may represent a unique cluster of related haplotypes, and different sets of haplotypes may occur in separate regions. As shown in the test networks, when only one SSL is available for each cluster, it becomes difficult to place new haplotypes within that cluster. Additionally, if a haplotype is only present in a single population, it is difficult to accurately place the haplotype within the network phylogeny. At large scales, it would be advisable to create artificial pools by pooling DNA from individuals from multiple populations located across the entire range. Notably, these pools should not include the samples used for SSLs, as those haplotypes are already known, and should contain samples at differing concentrations; the purpose of these pools is to help resolve the identity of any new haplotypes inferred by CallHap. Sequencing more than one SSL per population should also be considered in these cases. Sequencing multiple populations per region will also help resolve topologies and haplotype frequencies when the distance between populations within each region occurs at a small scale, and sampled regions occur at a large scale.

One final complication is that the true scale of a project may not become evident until after starting data analysis. For example, when the *L. californica* experiment was designed, the hypothesized dispersal range was greater than the distance between populations. However, after sequencing, it turned out that seed dispersal in *L. californica* much more limited than anticipated. In retrospect, creating artificial pools to help resolve the network topology would have been beneficial.

Pooling and Pooled library size – Many pool-seq protocols pool samples before DNA extraction (Kofler et al., 2012; Martins et al., 2014; Bélanger et al., 2016), but this may generate higher errors in SNP frequencies because equal amounts of tissue may not contain equal amounts of genomic DNA. In contrast, data for use in CallHap comes from libraries where DNA is extracted before being pooled to ensure equimolar proportions of DNA from each individual. While populations of any size could be analyzed, sequencing error, pipet volume, and DNA concentration limit the number of individuals that can be safely placed in a single PL and still give accurate resolution of haplotype frequencies. In addition, as the number of individuals in a PL increases, the frequency that represents a single individual starts to approach the level of error in the sequencing process. On the other hand, if too few individuals per population are used, some haplotypes present in the population may be missed. For example, if 10 individuals per population were used, any haplotype present at a frequency below 10% would likely go undetected. In the *L. californica* study, a sample size of 20 individuals per population was used; it provided reasonable accuracy in SNP frequency estimates while still capturing a good amount of the haplotype diversity present. More individuals per population could be used by

sequencing multiple pools per population, processing them as separate populations, and then combining the frequencies after running them through CallHap and before continuing with later population genetics or phylogeographic analysis.

Choosing a Reference Genome – CallHap assumes that SNPs detected by variant calling arise from closely related haplotypes. Because of this, the CallHap pipeline requires that all libraries be aligned to a single reference genome. Since the genome used will have a large influence on the number and quality of SNPs generated, genome selection is an important aspect of any study using CallHap.

In choosing a reference genome to use for CallHap analysis, preference should be given to conspecific references. If no such reference exists, one library of shotgun sequencing should be run; this library can be used to create a *de novo* reference genome to which the other samples can be aligned. While a *de novo* can be created using captured cpDNA, the incomplete nature of the capture makes it more difficult to carry out the *de novo* assembly. If creating a *de novo* reference is infeasible, it may be possible to obtain limited results using a non-conspecific reference; in this case, the more closely-related the reference chloroplast genome is to the study species, the better. Limitations of interspecific references include the addition of artificial SNPs introduced due to alignment ambiguities that may be caused by fixed differences between the chloroplast genomes of the two species.

Minimum Read Depth Selection – Another important parameter is the minimum read depth required to consider a genomic position for analysis. I found that this value changes depending on the peculiarities of different species and sequencing runs; for *L.*

californica, the optimum read depth was around 600, while for *Ranunculus occidentalis* Nutt. (Ranunculaceae), the optimum minimum depth was found to be 300-400. To determine the optimum minimum depth, I ran the VCF filter multiple times with different depths, and counted the number of initial unique haplotypes each time. I then selected the optimum depth as the point where the number of haplotypes started to drop off (Fig. 7) or 300, whichever was higher. In general, minimum depth should be no less than 15 * the number of individuals in a pool (Sims et al., 2014).

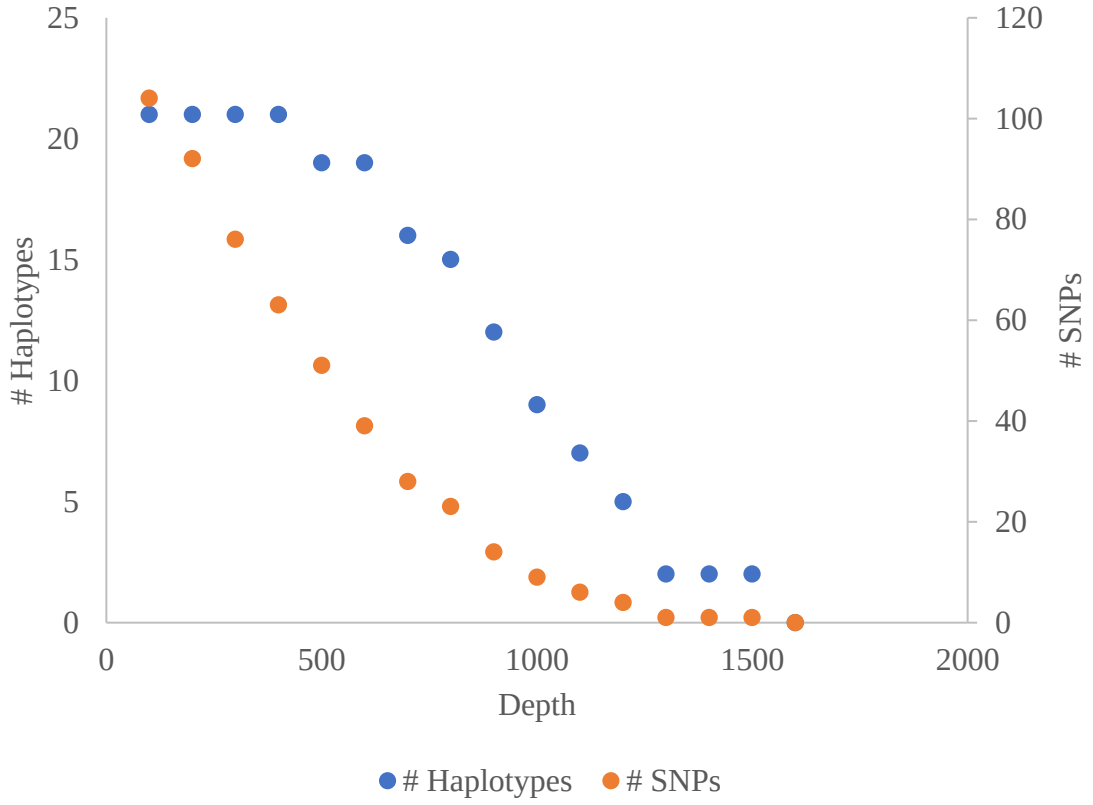


Fig. 7: Depth analysis for L. californica. The number of unique SSL haplotypes starts to drop off at around 600 depth.

Analysis of CallHap Outputs

Methods used for analysis of haplotype frequency data from CallHap will vary depending on the goals of the study. Population genetics studies utilizing nuclear genetic markers in diploid organisms typically use Wright's F_{ST} (Wright, 1949) or a similar analogue (G_{ST}, G'_{ST}, D_{ST}, etc.; Whitlock, 2011). However, F_{ST} is based on comparisons of observed and expected heterozygosity at different scales, and consequently is inappropriate for use with haplotype data. Instead, genetic distance measures that allow for variable ploidies and number of alleles per locus and are not reliant on measures of homo- or heterozygosity—such as Nei's Genetic Distance (Nei's D; Nei, 1973), Edwards chord distance (Cavalli-Sforza and Edwards, 1967; Edwards, 1971; Hartl et al., 1997), or Φ-statistics (Meirmans, 2006)—and haplotype genetic diversity measures (e.g. unbiased haplotype diversity; Gardner et al., 2015) should be used.

Methods such as Nei's D rely on calculations of the probability that the same combination of alleles will be found in two different populations, and consequently are more appropriate for small-scale studies. When no haplotypes are shared between two populations, Nei's D gives an infinite distance between those populations; such a pattern indicates that dispersal rates among the populations sampled are very low, and that the accumulation of local mutations is the primary factor contributing to the genetic structure of populations. Limited dispersal relative to the scale of sampling will lead to haplotypes within populations being more similar to each other than haplotypes in different populations, as can be seen in the *L. californica* data. In these cases, phylogeographic methods, the Edwards chord distance, or Φ-statistics will be more appropriate.

In phylogeographic studies or population genetics studies that are found to be more appropriate for phylogeographic analysis, methods such as Nested Clade Analysis (Templeton, 1998, 2009) or Approximate Bayesian Computation (Csilléry et al., 2010) should be used. These methods explain modern observations with predictions of population history events by comparing observed data to different modeled population histories.

Applications

The CallHap pipeline has the potential to creating a range of new opportunities for studies of cpDNA population structure, and allows for accurate and economical estimates of seed-mediated gene flow by allowing for the use of pooled population sequencing data for cpDNA and other haploid genetic material. Data for use in the CallHap pipeline comes from population-level sampling of haploid genomes, including plant chloroplast genomes (as presented in this paper), mitochondrial genomes, and prokaryotic bacterial genomes. Because CallHap assumes all generated haplotypes are closely related and requires that all libraries examined be aligned to a single reference genome; this protocol should not be used for microbiome and microbial community studies. Outputs generated by CallHap can be analyzed using a variety of methods, including Nei's genetic distance, Edwards chord distance, Φ-statistics, and a variety of phylogeographic analysis methods including Nested Clade Analysis and Approximate Bayesian Computation.

CallHap is available at https://github.com/cruzan-lab/CallHap.

References

BÉLANGER, S., P. ESTEVES, I. CLERMONT, M. JEAN, and F. BELZILE. 2016. Genotyping-by-sequencing on pooled samples and its use in measuring segregation bias during the course of androgenesis in barley. *The Plant Genome* 9: 0. Available at: https://dl.sciencesocieties.org/publications/tpg/abstracts/9/1/plantgenome2014.10.0073 [Accessed March 23, 2017].

CAIN, M.L., B.G. MILLIGAN, and A.E. STRAND. 2000. Long-distance seed dispersal in plant populations. *American Journal of Botany* 87: 1217–1227.

CAVALLI-SFORZA, L.L., and A W.F. EDWARDS. 1967. Phylogenetic analysis. Models and estimation procedures. *The American Journal of Human Genetics* 19: 233–257. Available at: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1706274&tool=pmcentrez&rendertype=abstract.

CORRIVEAU, J.L., and A.W. COLEMAN. 1988. Rapid Screening Method to Detect Potential Biparental Inheritance of Plastid DNA and Results for Over 200 Angiosperm Species. *American Journal of Botany* 75: 1443. Available at: http://www.researchgate.net/publication/250269704_Rapid_Screening_Method_to_Detect_Potential_Biparental_Inheritance_of_Plastid_DNA_and_Results_for_Over_200_Angiosperm_Species [Accessed March 17, 2015].

CSILLÉRY, K., M.G.B. BLUM, O.E. GAGGIOTTI, and O. FRANÇOIS. 2010. Approximate Bayesian Computation (ABC) in practice. *Trends in Ecology and Evolution* 25: 410–418.

EDWARDS, A.W.F. 1971. Distances between Populations on the Basis of Gene Frequencies. *Biometrics* 27: 873–881. Available at: http://www.jstor.org/stable/2528824 [Accessed March 24, 2017].

GARDNER, E.M., K.M. LARICCHIA, M. MURPHY, D. RAGONE, B.E. SCHEFFLER, S. SIMPSON, E.W. WILLIAMS, and N.J.C. ZEREGA. 2015. Chloroplast microsatellite markers for *Artocarpus* (Moraceae) developed from transcriptome sequences. *Applications in Plant Sciences* 3: apps.1500049. Available at: http://dx.doi.org/10.3732/apps.1500049 [Accessed March 28, 2017].

GARRISON, E., and G. MARTH. 2012. Haplotype-based variant detection from short-read sequencing. Available at: http://arxiv.org/abs/1207.3907 [Accessed January 25, 2017].

GASBARRA, D., S. KULATHINAL, M. PIRINEN, and M.J. SILLANPÄÄ. 2011. Estimating haplotype frequencies by combining data from large DNA pools with database information. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 8: 36–44. Available at: http://www.ncbi.nlm.nih.gov/pubmed/21071795.

GODOY, J.A., and P. JORDANO. 2001. Seed dispersal by animals: Exact identification of source trees with endocarp DNA microsatellites. *Molecular Ecology* 10: 2275–2283. Available at: http://doi.wiley.com/10.1046/j.0962-1083.2001.01342.x [Accessed March 22, 2017].

GORCHOV, D.L., F. CORNEJO, C. ASCORRA, and M. JARAMILLO. 1993. The role of seed dispersal in the natural regeneration of rain forest after strip-cutting in the peruvian

amazon. *Vegetatio* 107: 339–349. Available at: http://www.jstor.org/stable/20046318 [Accessed March 22, 2017].

HARTL, D.L., A.G. CLARK, and A.G. CLARK. 1997. Principles of population genetics. Sinauer associates Sunderland.

HOWE, H., and J. SMALLWOOD. 1982. Ecology of seed dispersal.

JOMBART, T. 2008. adegenet: a R package for the multivariate analysis of genetic markers. *Bioinformatics* 24: 1403–1405. Available at: https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btn129 [Accessed March 16, 2017].

JOSHI, N., and J. FASS. 2011. Sickle: A sliding-window, adaptive, quality-based trimming tool for FastQ files (Version 1.33) [Software]. *Available at https://github.com/najoshi/sickle.*2011.

KAYS, R., P.A. JANSEN, E.M.H. KNECHT, R. VOHWINKEL, and M. WIKELSKI. 2011. The effect of feeding time on dispersal of *Virola* seeds by toucans determined from GPS tracking and accelerometers. *Acta Oecologica* 37: 625–631. Available at: http://ac.els-cdn.com/S1146609X1100107X/1-s2.0-S1146609X1100107X-main.pdf?_tid=46e2ca44-0f43-11e7-86fa-00000aacb362&acdnat=1490216908_1d364bc5aadf4cdf2c6b851d2924832f [Accessed March 22, 2017].

KIRKPATRICK, B., C.S. ARMENDARIZ, R.M. KARP, and E. HALPERIN. 2007. HaploPool: Improving haplotype frequency estimation through DNA pools and phylogenetic modeling. *Bioinformatics* 23: 3048–3055.

KOFLER, R., A.J. BETANCOURT, and C. SCHLÖTTERER. 2012. Sequencing of Pooled DNA Samples (Pool-Seq) Uncovers Complex Dynamics of Transposable Element Insertions in *Drosophila melanogaster*. *PLoS Genetics* 8: e1002487. Available at: http://dx.plos.org/10.1371/journal.pgen.1002487.

KOFLER, R., R.V. PANDEY, and C. SCHLÖTTERER. 2011. PoPoolation2: Identifying differentiation between populations using sequencing of pooled DNA samples (Pool-Seq). *Bioinformatics* 27: 3435–3436.

KOLLMANN, J., and D. GOETZE. 1998. Notes on seed traps in terrestrial plant communities. *Flora* 193: 31–40. Available at: https://www.researchgate.net/profile/Johannes_Kollmann/publication/277709183_Notes_on_seed_traps_in_terrestrial_communities/links/559ab47608ae5d8f3937eaf3.pdf [Accessed March 22, 2017].

LI, B.B., J. MORRIS, and E.B. MARTIN. 2002. Model selection for partial least squares regression. *Chemometrics Intell. Lab. Syst.* 64: 79–89. Available at: http://ac.els-cdn.com/S0169743902000515/1-s2.0-S0169743902000515-main.pdf?_tid=cee90502-102e-11e7-b17f-00000aacb361&acdnat=1490318068_848534ed2af579297aa9d6b100601ef8 [Accessed March 23, 2017].

LI, H., and R. DURBIN. 2009. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*.

MARTIN, M. 2011. Cutadapt removes adapter sequences from high-throughput sequencing reads. *EMBnet.journal* 17: 10. Available at:

http://journal.embnet.org/index.php/embnetjournal/article/view/200 [Accessed November 14, 2016].

MARTINS, N.E., V.G. FARIA, V. NOLTE, C. SCHLÖTTERER, L. TEIXEIRA, É. SUCENA, and S. MAGALHÃES. 2014. Host adaptation to viruses relies on few genes with different cross-resistance properties. *Proceedings of the National Academy of Sciences of the United States of America* 111: 5938–43. Available at: http://www.ncbi.nlm.nih.gov/pubmed/24711428 [Accessed March 23, 2017].

MCKENNA, A., M. HANNA, E. BANKS, A. SIVACHENKO, K. CIBULSKIS, A. KERNYTSKY, K. GARIMELLA, ET AL. 2010. The genome analysis toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*.

MEIRMANS, P.G. 2006. Using the AMOVA framework to estimate a standardized genetic differentiation measure. *Evolution* 60: 2399–2402. Available at: http://dx.doi.org/10.1554/05-631.1 [Accessed March 27, 2017].

MOUISSIE, A.M., C.E.J. VAN DER VEEN, G.F. (CISKA) VEEN, and R. VAN DIGGELEN. 2005. Ecological correlates of seed survival after ingestion by Fallow Deer. *Functional Ecology* 19: 284–290. Available at: http://doi.wiley.com/10.1111/j.0269-8463.2005.00955.x [Accessed May 4, 2015].

NATHAN, R., and H.C. MULLER-LANDAU. 2000. Spatial patterns of seed dispersal, their determinants and consequences for recruitment. *Trends in Ecology and Evolution* 15: 278–285. Available at: http://linkinghub.elsevier.com/retrieve/pii/S0169534700018747 [Accessed March 22, 2017].

NEI, M. 1973. Analysis of gene diversity in subdivided populations. *Proceedings of the National Academy of Sciences of the United States of America* 70: 3321–3323.

PALMER, J.D. 1987. Chloroplast DNA evolution and biosystematic uses of chloroplast DNA variation. *American Naturalist* 130: S6–S29. Available at: http://www.jstor.org/stable/2461917 [Accessed March 22, 2017].

PE'ER, I., and J.S. BECKMANN. 2003. Resolution of haplotypes and haplotype frequencies from SNP genotypes of pooled samples. *Proceedings of the seventh annual international conference on Computational molecular biology - RECOMB '03*237–246. Available at: http://dl.acm.org/citation.cfm?id=640075.640107.

PRITCHARD, J.K., M. STEPHENS, and P. DONNELLY. 2000. Inference of population structure using multilocus genotype data. *Genetics* 155: 945–959. Available at: http:/ [Accessed March 16, 2017].

RAJ, A., M. STEPHENS, and J.K. PRITCHARD. 2014. FastSTRUCTURE: Variational inference of population structure in large SNP data sets. *Genetics* 197: 573–589. Available at: http://web.stanford.edu/group/pritchardlab/publications/pdfs/Raj14 [Accessed March 16, 2017].

SBONER, A., X. MU, D. GREENBAUM, R.K. AUERBACH, M.B. GERSTEIN, M. METZKER, E. MARDIS, ET AL. 2011. The real cost of sequencing: higher than you think! *Genome Biology* 12: 125. Available at: http://genomebiology.com/2011/12/8/125 [Accessed March 22, 2017].

SCHLÖTTERER, C., R. TOBLER, R. KOFLER, and V. NOLTE. 2014. Sequencing pools of individuals — mining genome-wide polymorphism data without big funding. *Nature*

Reviews Genetics 15: 749–763. Available at: http://www.nature.com/doifinder/10.1038/nrg3803 [Accessed March 23, 2017].

SHAM, P., J.S. BADER, I. CRAIG, M. O'DONOVAN, and M. OWEN. 2002. DNA Pooling: a tool for large-scale association studies. *Nat Rev Genet* 3: 862–871. Available at: http://www.nature.com/nrg/journal/v3/n11/pdf/nrg930.pdf [Accessed March 22, 2017].

SIMS, D., I. SUDBERY, N.E. ILOTT, A. HEGER, and C.P. PONTING. 2014. Sequencing depth and coverage: key considerations in genomic analyses. *Nature reviews. Genetics* 15: 121–32. Available at: http://dx.doi.org/10.1038/nrg3642 [Accessed July 11, 2014].

SLATKIN, M. 1987. Gene Flow and the Geographic Structure of Natural Populations. *Science* 236: 787–792. Available at: http://science.sciencemag.org/content/236/4803/787 [Accessed March 22, 2017].

STULL, G.W., M.J. MOORE, V.S. MANDALA, N. A DOUGLAS, H.-R. KATES, X. QI, S.F. BROCKINGTON, ET AL. 2013. A targeted enrichment strategy for massively parallel sequencing of Angiosperm plastid genomes. *Applications in Plant Sciences* 1: 1–7. Available at: http://www.bioone.org/doi/abs/10.3732/apps.1200497.

TEMPLETON, A.R. 1998. Nested clade analyses of phylogeographic data: Testing hypotheses about gene flow and population history. *Molecular Ecology* 7: 381–397. Available at: http://doi.wiley.com/10.1046/j.1365-294x.1998.00308.x [Accessed February 7, 2017].

TEMPLETON, A.R. 2009. Statistical hypothesis testing in intraspecific phylogeography: Nested clade phylogeographical analysis vs. approximate Bayesian computation.

Molecular Ecology 18: 319–331.

TEMPLETON, A.R., K.A. CRANDALL, and C.F. SING. 1992. A cladistic analysis of phenotypic associations with haplotypes inferred from restriction endonuclease mapping and DNA sequence data. III. Cladogram estimation. *Genetics* 132: 619–633. Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1205162/pdf/ge1322619.pdf [Accessed March 28, 2017].

THORVALDSDÓTTIR, H., J.T. ROBINSON, and J.P. MESIROV. 2013. Integrative Genomics Viewer (IGV): High-performance genomics data visualization and exploration. *Briefings in Bioinformatics*.

TRAKHTENBROT, A., R. NATHAN, G. PERRY, and D.M. RICHARDSON. 2005. The importance of long-distance dispersal in biodiversity conservation. *Diversity and Distributions* 11: 173–181. Available at: http://doi.wiley.com/10.1111/j.1366-9516.2005.00156.x [Accessed February 7, 2017].

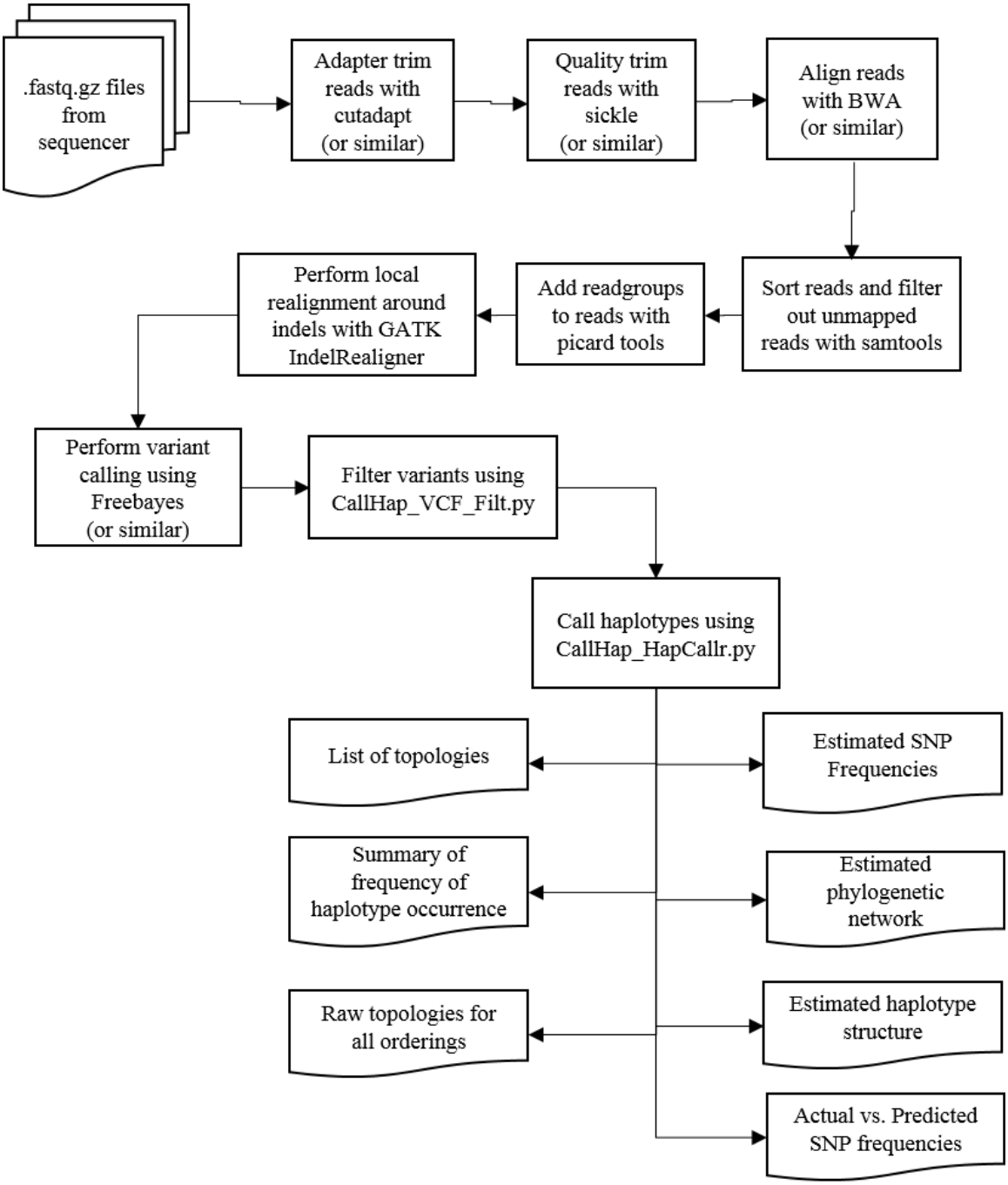
WALKER, J.F., M.J. ZANIS, and N.C. EMERY. 2014. Comparative analysis of complete chloroplast genome sequence and inversion variation in *Lasthenia burkei* (Madieae, Asteraceae). *American journal of botany* 101: 722–9. Available at: http://www.amjbot.org/content/101/4/722.long [Accessed April 20, 2015].

WHITLOCK, M.C. 2011. G'ST and D do not replace FST. *Molecular Ecology* 20: 1083–1091. Available at: http://doi.wiley.com/10.1111/j.1365-294X.2010.04996.x.

WILLSON, M.F. 1993. Dispersal mode, seed shadows, and colonization patterns. *Vegetatio* 107–108: 261–280.

WRIGHT, S. 1949. The genetical structure of populations. *Annals of Eugenics* 15: 323–354. Available at: http://doi.wiley.com/10.1111/j.1469-1809.1949.tb02451.x [Accessed December 1, 2014].

Appendix A: CallHap Bioinformatics Pipeline Overview

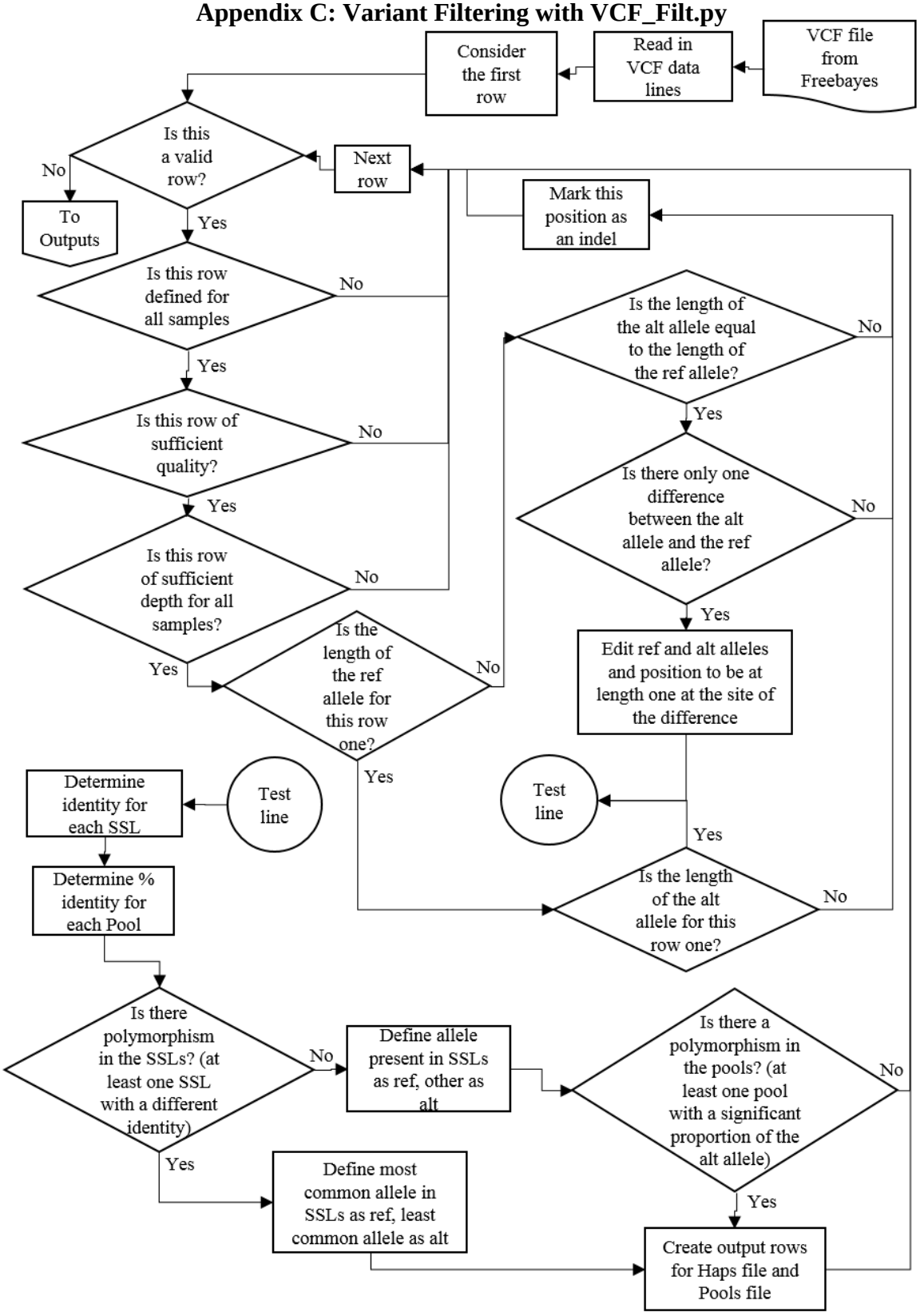


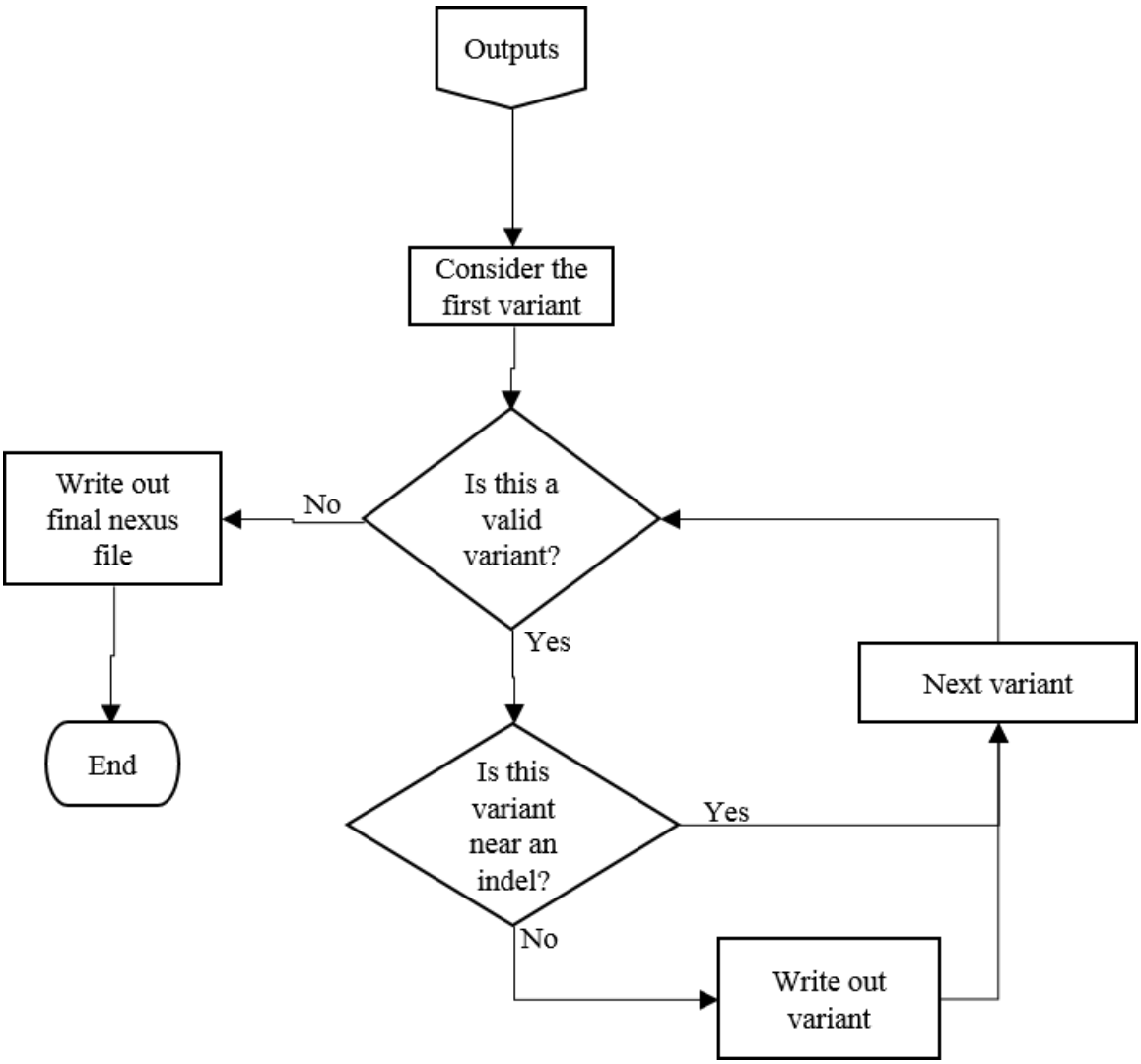
Appendix B: Capture Array Creation

A MYbaits-3 custom cpDNA capture array from MYcroarray (MYcroarray, Ann Arbor, MI, USA) was created to help isolate cpDNA. During capture array creation, 120mer baits were constructed with a ~2x flexible tiling density. Any baits with 10 or less mismatches between them were collapsed into a single bait. In total, the capture array contained 55,409 baits.

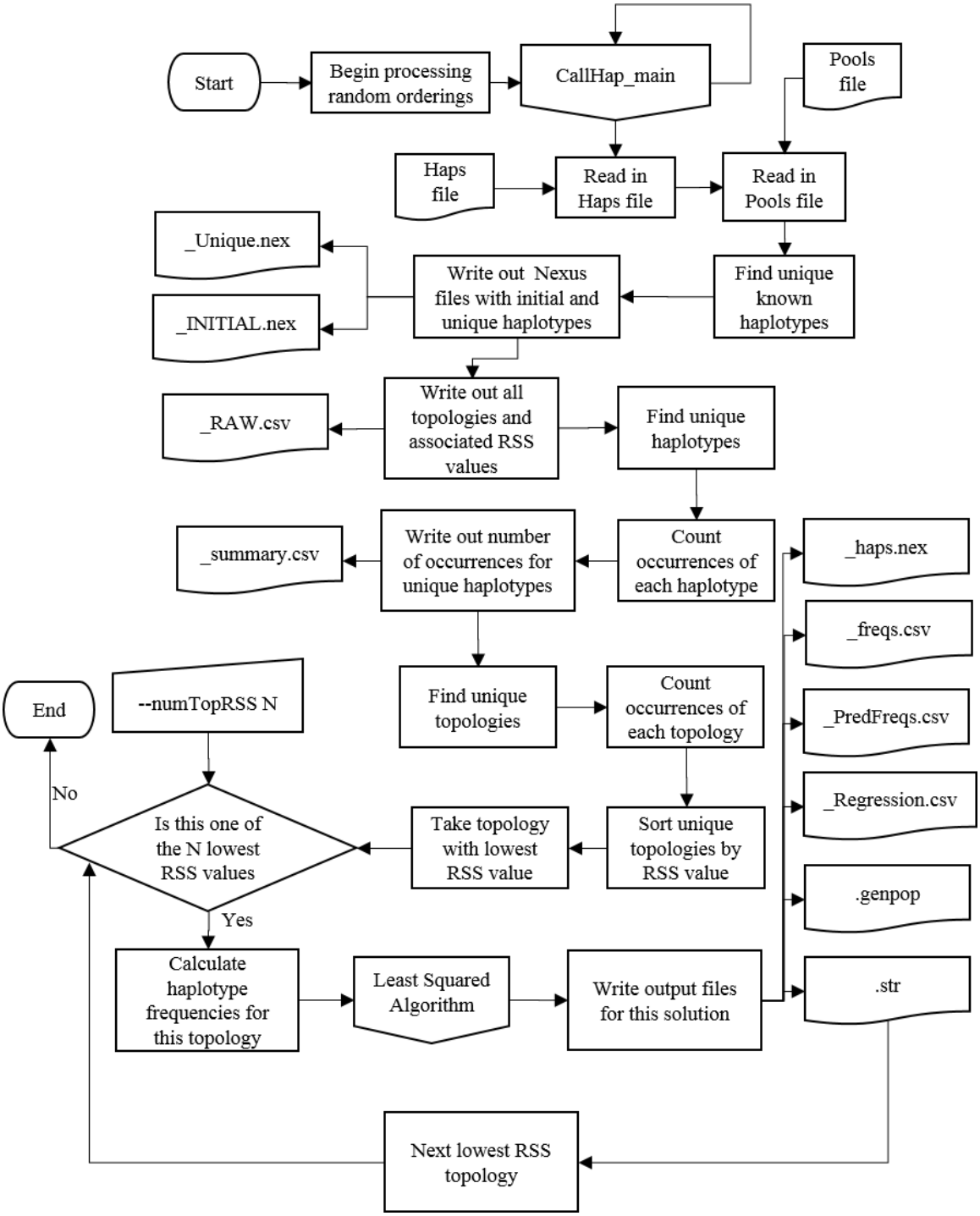
Species	Source
Achillea millefolium	Partial *De Novo*
Achyrachaena mollis	Partial *De Novo*
Arabis alpa	NCBI: NC_023367.1
Brachypodium distachyon	NCBI: NC_011032.1
Camassia quamash	Partial *De Novo*
Chrysanthemum indicum	NCBI: NC_020320.1
Cryptantha torreyana	NCBI: KP096524.1
Danthonia californica	NCBI: NC_025232.1
Danthonia californica	Partial *De Novo*
Eriophyllum lanatum	Partial *De Novo*
Eustrephus latifolius	NCBI: NC_025305.1
Festuca arundinacea	NCBI: NC_011713_2
Festuca roemeri	Partial *De Novo*
Fragaria vesca	NCBI: NC_015206.1
Lactusa sativa	NCBI: NC_007578.1
Lasthenia burkei	NCBI: KM360047.1
Lomatium utriculatum	Partial *De Novo*
Lupinus albus	NCBI: NC_026681.1
Lupinus bicolor	Partial *De Novo*
Nama carnosum	Private communication with Gregory Stull
Nicotina undulata	NCBI: NC_016068.1
Petroselinium crispum	NCBI: HM596073.1
Quercus aliena	NCBI: KP301144.1
Ranunculus austro-oreganus	Partial *De Novo*
Ranunculus macranthus	NCBI: NC_008796.1
Ranunculus occidentalis	Partial *De Novo*
Salvia miltiorrhiza	NCBI: NC_020431.1
Hibiscus syriacus	NCBI: NC_026909.1
Oenothera biennis	NCBI: NC_010361.1
Lonicera japonica	NCBI: NC_026839.1
Lilium superbum	NCBI: NC_026787.1
Primula poissonii	NCBI: NC_024543.1
Liquidambar formosana	NCBI: NC_023092.1

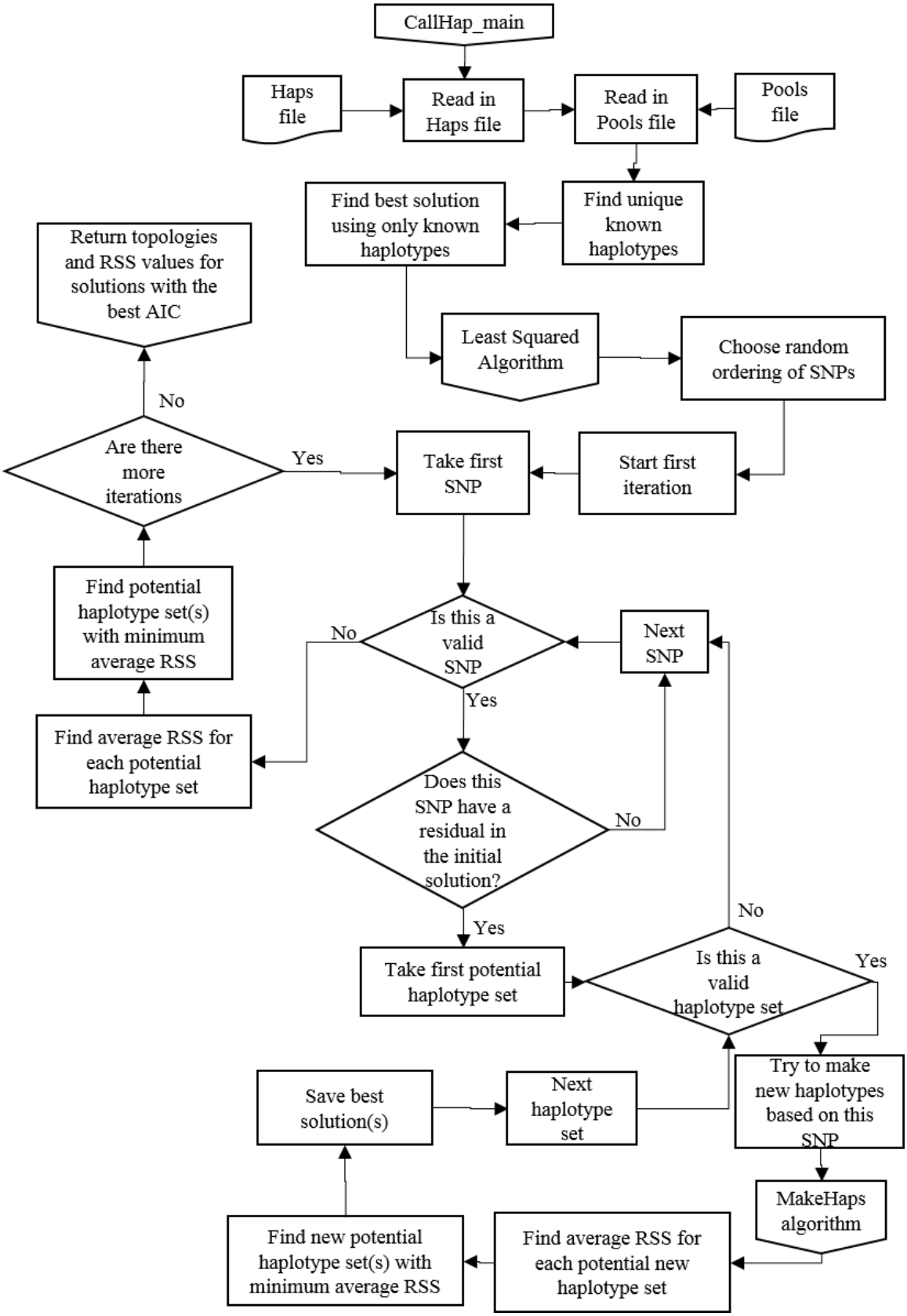
Appendix C: Variant Filtering with VCF_Filt.py



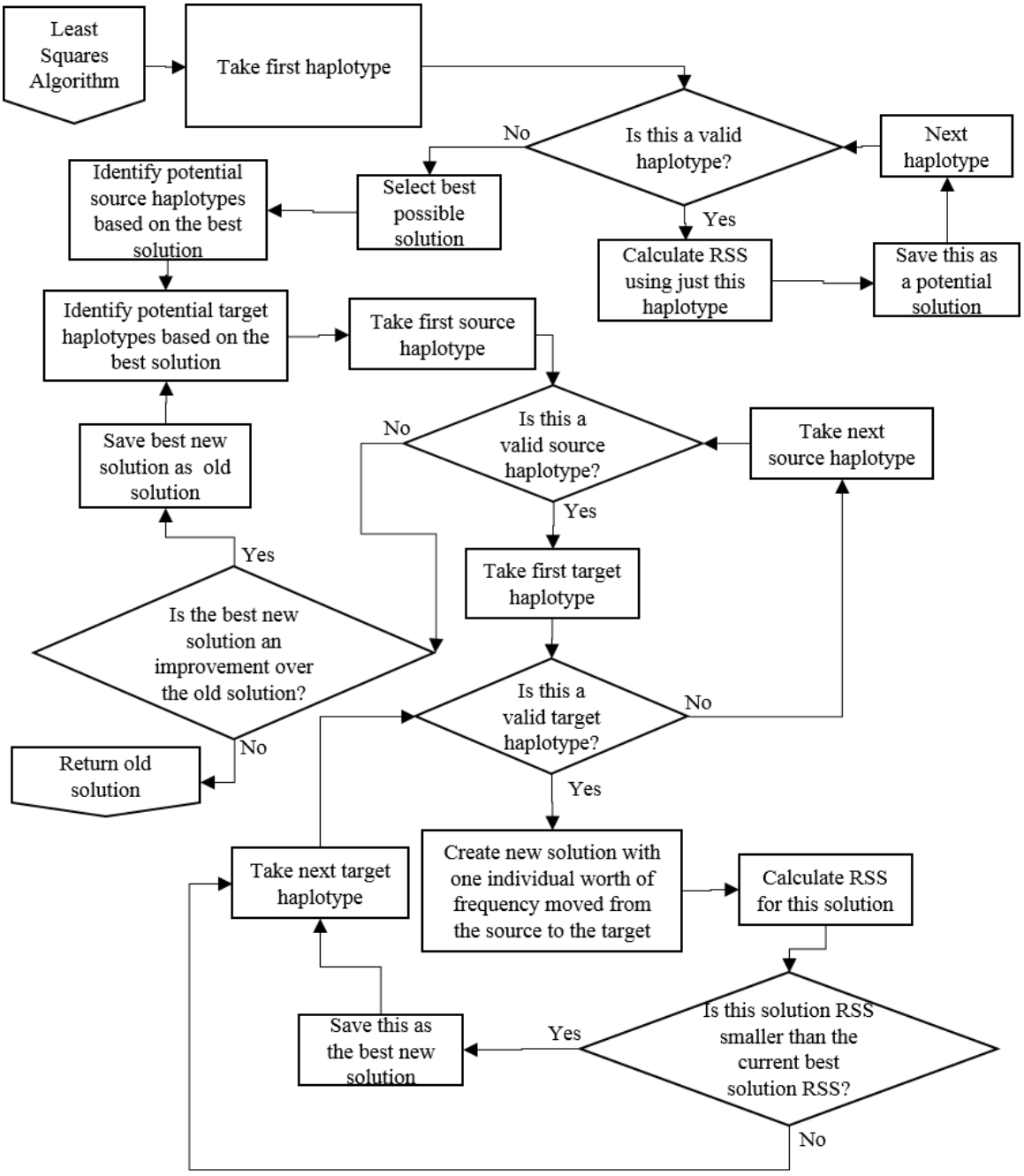

Outputs
Consider the first variant
Is this a valid variant?
No
Write out final nexus file
End
Yes
Is this variant near an indel?
Yes
Next variant
No
Write out variant

Appendix D: Overall haplotype and frequency estimation program (HapCallr.py)

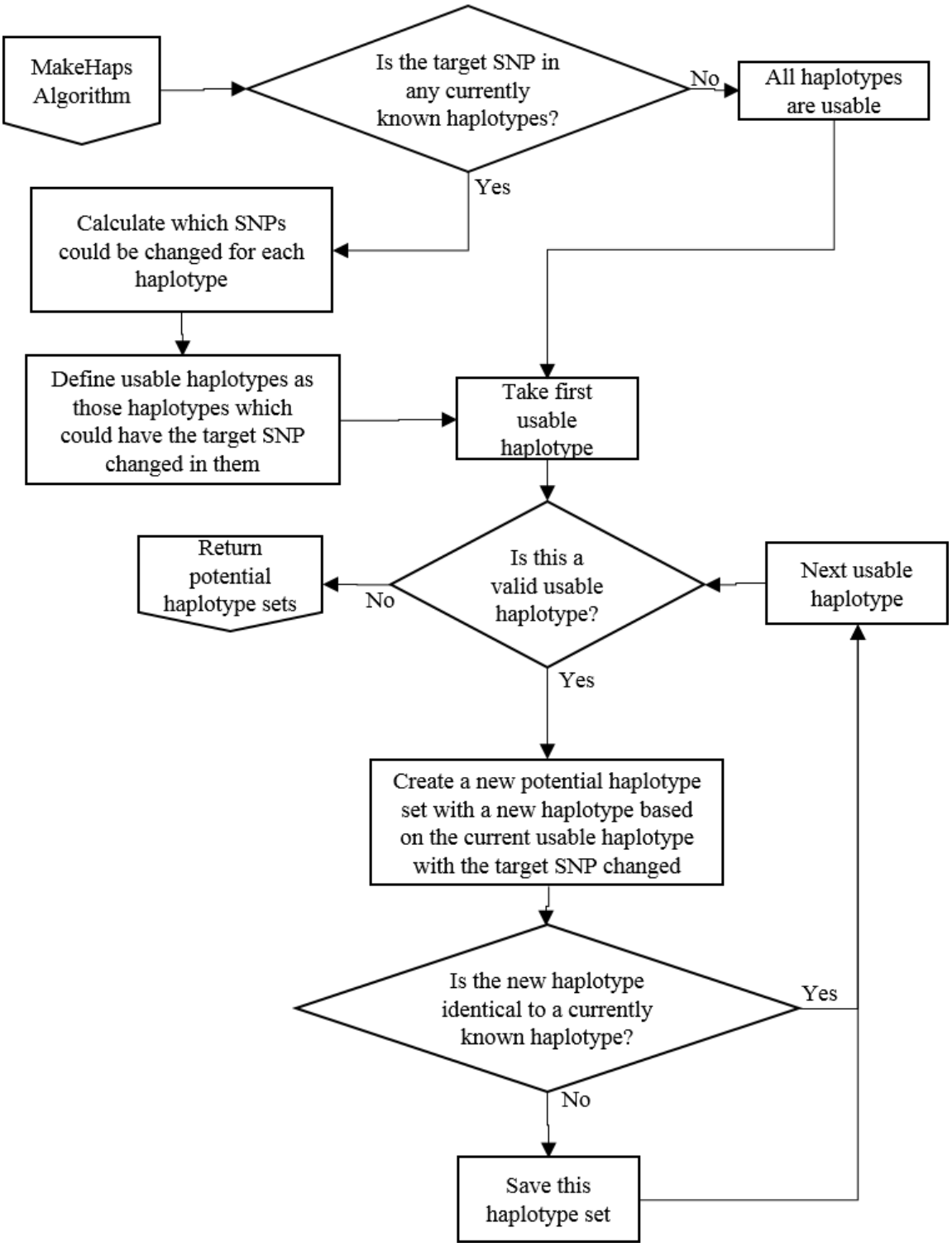



CallHap_main
Haps file
Read in Haps file
Read in Pools file
Pools file
Find unique known haplotypes
Find best solution using only known haplotypes
Return topologies and RSS values for solutions with the best AIC
Least Squared Algorithm
Choose random ordering of SNPs
No
Are there more iterations
Yes
Take first SNP
Start first iteration
Find potential haplotype set(s) with minimum average RSS
No
Is this a valid SNP
Next SNP
Yes
Find average RSS for each potential haplotype set
Does this SNP have a residual in the initial solution?
No
Yes
No
Take first potential haplotype set
Is this a valid haplotype set
Yes
Try to make new haplotypes based on this SNP
Save best solution(s)
Next haplotype set
MakeHaps algorithm
Find new potential haplotype set(s) with minimum average RSS
Find average RSS for each potential new haplotype set

Appendix E: CallHap Least Squares Algorithm



Appendix F: CallHap Haplotype Creation Algorithm



Appendix G: CallHap Manual

CallHap: A Pipeline for Analysis of Pooled Whole-genome Haplotypes

Last edited: 04/28/2017 By: Jessica Persinger

Licensing information

With the exception of the Genome Analysis Toolkit, all programs are freely available under either the Gnu Public License or the MIT License. The Genome Analysis Toolkit is free for non-commercial use; other use should contact the Broad Institute at softwarelicensing@broadinstitute.org. Python and bash scripts for the CallHap pipeline are available at https://github.com/cruzan-lab/CallHap.

Introduction

CallHap is a pipeline designed for the analysis of pooled haplotype data. It depends on the presence of two types of sequencing libraries; either single sample libraries (SSLs) or pooled libraries (Pool). Ideally, a Pool should contain equimolar genetic material from 20 individuals, and one of those individuals should be prepared separately as a SSL. This pipeline picks up following sequencing on an Illumina HiSeq or similar high-throughput sequencer.

Requirements

- A LINUX/UNIX/MacOS system with the following programs installed:
  - Cutadapt (http://cutadapt.readthedocs.io/en/stable/index.html)
  - Sickle (http://bioinformatics.ucdavis.edu/research-computing/software/)
  - BWA (http://bio-bwa.sourceforge.net/)
  - Samtools (http://samtools.sourceforge.net/)
  - PicardTools (https://broadinstitute.github.io/picard/)
  - GATK (https://software.broadinstitute.org/gatk/)
  - Freebayes (https://github.com/ekg/freebayes)
  - Python 2.7x (https://www.python.org/) with NumPy (http://www.numpy.org/)
  - Java Development Kit

Contents

Quick Start

Setup:

program-config.sh:

Edit program-config.sh so that each of the variables is set to the absolute path of the program in question.

Reference Preparation:

Obtain a reference genome (in FASTA format) for your species of interest (or closely related other species), and prepare it for use by using the following commands:

```
$ bwa index {reference}.fasta
$ samtools faidx {reference}.fasta
$ java -jar /path/to/picardtools/picard.jar \
  CreateSequenceDictionary \
  R={reference}.fasta \
  O={reference}.dict
```

Preprocessing:

Note that there are two basic processing pipelines provided; one with automated trimming (CallHap_Preproc_0.01.23.sh) and one without automated trimming (CallHap_Preproc_NoTrimming_0.01.23.sh). It is strongly suggested that at least a few (2-5) samples per flow cell be run manually (one step at a time), at least through trimming for quality control and to see if those samples need any additional trimming beyond the basic trimming steps (adapter and quality trimming). If you are doing trimming separately, be sure to use the locations of the trimmed files in the preconfig instead of the locations of the raw files.

Create a preconfig file in Excel with the following columns:

- Read1File
- Read2File
- RGLB
- RGSM
- RGPU
- Mode
- Reference

Each row should represent one sequencing library (SSL or Pooled).

- Read1File and Read2File should give the absolute path to the locations of the raw data for the Read 1 and Read 2 files (in the case of single end data, give the file location under Read1File, and put a period (.) for Read2File).
- RGLB should be some identifier for the library (e.g. library number).
- RGSM should be a sample name, preferably indicating the species of the library, the location the sample came from, and whether the sample is a SSL or Pool (Example: SpenamLocS#SSL, SpenamLocS#Pool).

- RGPU should indicate the barcoding used for this library during library prep (Example: ATTACTCG-TATAGCCT).
- Mode should be one of se (single-end) or pe (paired-end).
- Reference should indicate the reference genome you would like this library aligned to.

If all samples are of the same species, the reference genomes for all libraries should be the same.

Save the preconfig file as a .csv. Convert it to a config file using:

```
$ python /path/to/CallHap/CallHap_ConfigCreator.py \
  --input preconfig.csv \
  --adapt1 {SequencingAdapter} \
  --adapt2 {SequencingAdapter} \
  --sequencer {Sequencer used to produce data} \
  --minBaseQual 30 \
  --minReadQual 30 \
  --runID {Identifier for this run}
```

This will output a .sh file with the run ID as the name (for example, of you put --runID {runID}, the file would be called runID.sh)
Then use the following command to run the rest of the pre-processing (replacing the script name if you did trimming separately):

```
$ bash /path/to/CallHap/CallHap_Preproc_0.01.23.sh \
  program-config.sh {runID}.sh
```

SNP Calling:
Set up an input list of files using:

```
$ ls -1 /path/to/files/*SSL*.rg.ra.bam > {RunID}.txt
$ ls -1 /path/to/files/*Pool*.rg.ra.bam >> {RunID}.txt
```

Call FreeBayes using:

```
$ /path/to/freebayes/freebayes -L {RunID}.txt \
  -p 1 -f /path/to/reference/{reference}.fasta \
  -v {RunID}_SNPs.vcf --use-best-n-alleles 2 \
  --min-repeat-entropy 1 --no-partial-observations \
  --min-alternate-fraction {0.05}
```

`--min-alternate-fraction`	Should be set to 1/poolsize or lower.

This step may take a while, and while running, may look like it isn't doing anything

SNP Filtering:
SNP filtering is accomplished by use of a custom python script, which can be run with the command:

```
$ python /path/to/CallHap/CallHap_VCF_Filt.py \
  -i {RunID}_SNPs.vcf -o {RunID}_d{600}q{20}_Haps.vcf \
  -O {RunID}_d{600}q{20}_Pools.vcf -n <number of SSLs> \
  -N <number of Pools> -d {600} -q {20} -p {20}
```

You may need to trim off one or more columns from the VCF file if one sample was not called at a majority of positions; if a single sample is not called at a particular position, the variant at that position will be discarded. To determine if a column needs to be removed, look at your VCF file in Excel, and see if there are any columns that are periods (".") for the majority of rows. Removing the column can also be done in Excel, but you need to be careful because Excel likes to add quotes when it saves files with commas in the cells, as do most spreadsheet editors I've found.

Haplotype Calling:
Before running this step, check how many cores are available on the system you're using with htop. Make sure you don't overload the system you're working on; don't set -t to higher than the number of available cores, and don't take up all the cores on the machine.

Haplotype calling can be run using:

```
$ python /path/to/CallHap/CallHap_HapCallr.py \
  --inputHaps {RunID}_d250q20_Haps.vcf \
  --inputFreqs {RunID}_d250q20_Pools.vcf \
  -o {RunID} -p 20 -t 5 -l 2 --numRandom 100 \
  --numTopRSS 3 --genpop --structure
```

This program generates four to six output file per solution output (within the minimum number of RSS values):

- A NEXUS file (RunID_solNum_haps.nex) for network phylogeny creation; PopART (http://popart.otago.ac.nz/index.shtml) works fairly well. I've been using the TCS algorithm.
- A VCF file (RunID_solNum_PredFreqs.vcf) containing the estimated SNP frequencies based on the estimated haplotype frequencies, and the per-SNP average residuals in the INFO field
- A CSV file (RunID_solNum_freqs.csv) containing the per-pool haplotype frequencies and RSS values for each pool.
- A CSV file (RunID_solNum_Regression.csv) containing paired observed and predicted SNP frequencies from the Least Squared algorithm.

- (Optional): A Structure formatted file (RunID_solNum_iterNum.str) containing the expanded haplotype frequencies
- (Optional): A Genpop file containing the haplotype frequencies for use in Adigenet.

In addition, outputs are generated describing the original haplotypes network (RunID_Initial.nex), the unique haplotypes network (RunID_Unique.nex), raw topologies observed from each random order (RunID_RAW.csv), the frequency of each unique topology generated (RunID_topologies.csv), the frequency of occurrence for each haplotype found in any random order (RunID_summary.csv).
In terms of population-genetics analysis, haplotypes should be treated as independent alleles at a single locus.

Detailed Instructions

Adapter/Quality trimming:

Adapter and quality trimming should be performed before any other step in the pipeline. This ensures better read alignment and higher quality of the final data. The automated pipeline uses cutadapt for adapter trimming and sickle for quality trimming; however, you can use other trimming programs if so desired.

Cutadapt is available at http://cutadapt.readthedocs.io/en/stable/index.html under the MIT License and can be run using:

```
$ /path/to/cutadapt -a {inAdapter1} -A {inAdapter2} \
  -o {output_read_1}_at.fastq.gz \
  -p {output_read_2}_at.fastq.gz \
  {input read 1}.fastq.gz {input read 2}.fastq.gz
```

for paired-end reads or

```
$ /path/to/cutadapt -a {inAdapter1} \
  -o {output read 1} at.fastq.gz {input read 1}.fastq.gz
```

for single-end reads.

If you aren't certain what adapter sequence you have, running FastQC (freely available at http://www.bioinformatics.babraham.ac.uk/projects/fastqc/ under GPLv3) may help determine what adapters are present. Otherwise, consult your library preparation protocol.

While cutadapt can also do quality trimming (using the -q option), or remove a fixed number of bases (using the -u option), the default pipeline uses a second program, (sickle) for quality trimming. Sickle is available at http://bioinformatics.ucdavis.edu/research-computing/software/ under the MIT License and can be run with

```
$ /path/to/sickle pe -f {output_read_1}_at.fastq.gz \
  -r {output_read_2}_at.fastq.gz -o \
  {output_read_1}ut_at_qt.fastq.gz -p \
  {output_read_2}_at_qt.fastq.gz -t sanger -s \
  {SampleName} extras.fastq.gz -q {minBaseQuality} -g
```

for paired-end reads or

```
$ /path/to/sickle se -f {output_read_1}_at.fastq.gz \
  -o {output_read_1}_at_qt.fastq.gz -t sanger \
  -q {minBaseQuality} -g
```

for single-end reads.

For more details on these programs, consult their respective manuals.

Following trimming, it is recommended that at least 2-5 samples per flow cell be quality-checked using FastQC. For this pipeline, check that there are almost no remaining adapters of any type in the AdapterContent page of the report and that you are satisfied with the quality scores in the Per base sequence quality section and the base percentages in the Per base sequence content section.

Note that FastQC will generate output files in the same directory as the input files.

Read alignment:
The automated pipeline uses BWA-mem to align reads with default options. BWA can be obtained from http://bio-bwa.sourceforge.net/ under GPLv3, and can be run using:

```
$ /path/to/bwa mem -M {reference}.fasta \
  {output_read_1}_at_qt.fastq.gz \
  {output_read_2}_at_qt.fastq.gz > \
  {SampleName}.sam
```

for paired-end reads or

```
$ /path/to/bwa mem -M {reference}.fasta \
  {output_read_1}_at_qt.fastq.gz > {SampleName}.sam
```

for single-end reads.
After alignment, the file is converted to a bam file:

```
$ /path/to/samtools view -Sbu -F 4 {SampleName}.sam | \
  /path/to/samtools sort - {SampleName}.sort
```

Index the bam file:

```
$ /path/to/samtools index {SampleName}.sort.bam
```

At this time, any unaligned reads are also removed.
Samtools can be obtained from http://www.htslib.org/.

Readgroup Creation:
PicardTools is used to add readgroups to the files. These are a requirement for local realignment with GATK, and for SNP calling with FreeBayes. For later analysis, it is useful if each sample have a different sample name (RGSM) and readgroup ID (RGID),

since Freebayes (our SNP caller) uses the readgroup ID to differentiate samples. I used the library number as the readgroup ID.

PicardTools is available at https://broadinstitute.github.io/picard/, and can be run using

```
$ java -jar /path/to/picard AddOrReplaceReadGroups \
  INPUT={SampleName}.sort.bam \
  OUTPUT={SampleName}.sort.rg.bam \
  RGID={ReadGroupID} \
  RGLB={ReadGroupLibrary} \
  RBPL={ReadGroupSequencingPlatform} \
  RGPU={ReadGroupRunBarcode} \
  RGSM={ReadGroupSampleName} \
  CREATE_INDEX=true
```

RGLB, RBPL, RGPU, and RGSM are required for this tool to run.
RGID needs to be different for each library.

Local Realignment:
Local realignment is carried out using the Genome Analysis Toolkit (GATK, available at https://software.broadinstitute.org/gatk/). The first step in this process is to locate targets for local realignment using:

```
$ java -jar /path/to/GATK -T RealignerTargetCreator \
  -R {reference}.fasta \
  -I {SampleName}.sort.rg.bam \
  -o {SampleName}.sort.rg.intervals
```

Following this, local realignment can be run using:

```
$ java -jar /path/to/GATK -T IndelRealigner \
  -R {reference}.fasta \
  -I {SampleName}.sort.rg.bam \
  -targetIntervals {SampleName}.sort.rg.intervals \
  -o {SampleName}.sort.rg.ra.bam \
  -dt NONE \
  --maxReadsForRealignment 200000
```

SNP Calling:
Set up an input list of files using:

```
$ ls -1 /path/to/files/*SSL*.rg.ra.bam > {RunID}.txt
$ ls -1 /path/to/files/*Pool*.rg.ra.bam >> {RunID}.txt
```

Or whatever identifier you used to differentiate PLs and SSLs. The important thing is that this file list all SSLs, followed by all PLs.

Call FreeBayes using:

```
$ /path/to/freebayes/freebayes -L {RunID}.txt \
  -p 1 -f /path/to/reference/{reference}.fasta \
  -v {RunID}_SNPs.vcf --use-best-n-alleles 2 \
  --min-repeat-entropy 1 --no-partial-observations \
  --min-alternate-fraction 0.05
```

`--min-alternate-fraction`	Should be set to 1/poolsize or lower.

This step may take a while, and while running, may look like it isn't doing anything. FreeBayes can be found at https://github.com/ekg/freebayes.

SNP Filtering:
Before running SNP filtering, it may be necessary to trim off one or more columns from the VCF file if one sample was not called at a majority of positions; if a single sample is not called at a particular position, the variant at that position will be discarded, so a single sample uncalled (or at low depth) at a majority of positions can result in no data making it through the filtering step. To determine if a column needs to be removed, look at your VCF file in Excel, and see if there are any columns that are periods (".") for the majority of rows. Removing the column can also be done in Excel, but you need to be careful because Excel likes to add quotes when it saves files with commas in the cells, as do most spreadsheet editors I've found.

If desired, sample depth can be assessed using the GATK DepthOfCoverage tool (see https://software.broadinstitute.org/gatk/documentation/tooldocs/org_broadinstitute_gatk_tools_walkers_coverage_DepthOfCoverage.php for instructions). This tool takes a similar amount of time to SNP calling.

SNP filtering is accomplished by use of a custom python script, which can be run with the command:

```
$ python /path/to/CallHap/CallHap_VCF_Filt.py \
  -i {RunID}_SNPs.vcf -o {RunID}_d600q20_Haps.vcf \
  -O {RunID}_d600q20_Pools.vcf -n <number of SSLs> \
  -N <number of Pools> -d 600 -q 20 -p 20
```

`-i`	The input VCF file from FreeBayes
`-o`	The output haplotypes file, containing haplotypes found in the SSLs

`-O`	The output Pool SNP frequencies file, containing frequency of the more common allele in each pool
`-n`	The number of SSLs in the input file
`-N`	The number of Pools in the input file
`-d, --minDepth`	This option sets the minimum read depth that must be present at a position in ALL libraries in order for that position to be considered as a variant. It should be set based on the number of individuals in a PL. For haploid sequence, a depth of 15 per individual in the pool is recommended (Sims et al., 2014), so that for a pool of 20 individuals, a depth of 300 is required at a site to be able to call variants.
`-q, --minQual`	Controls the minimum PHRED-scaled variant quality needed to use a variant. Mostly useful for filtering out super-low quality variants, but can be set higher as necessary. -p is the number of individuals in each pool.
`--minCallPrev`	Controls the maximum allowable error in SSLs for a variant to be called. It can range from 1 (all reads in each SSL need to have the same identity) to 0.5 (Up to half the reads in a SSL can have a different identity). At a setting of 1, some real SNPs could be removed based on unavoidable errors in the SSLs, while at a setting of 0.5, confidence in the identity call for SSLs, and thus in the identity of haplotypes, will be significantly decreased. I set this parameter at a default of 0.9, to allow for some sequencing error in the SSLs while still maintaining a high accuracy of SSL identity calls.
`--minSnpPrev`	Coupled with poolSize, this option controls how much of a PL must be the alternate identity for a SNP to be at that position when there is no variation in the SSLs. The value is a positive floating-point decimal, which gets multiplied by 1/poolSize to yield the proportion of reads that must be of a different identity in a PL to yield a variant. At a value of zero, all positions would be called as variants if there was any variation in a PL. I set this at a default value of 0.75 in order to allow for some error in low-frequency haplotypes, while removing the majority of low-frequency sequencing errors from consideration.
`--indelDist`	How far away from indels a variant should be for use. IndelDist takes an integer value greater than 0; at a value of 0, distance from an indel will not be considered as a filter. I set this at a relatively conservative value of 100 (the length of my raw sequencing reads) as being the maximum distance at which the presence of an indel could have any effect on variant discovery.

It is recommended to run this program with different sets of parameters to determine what the optimum parameters will be for a particular run.

Haplotype Calling:
Before running this step, check how many cores are available on the system you're using with htop. Make sure you don't overload the system you're working on; don't set -t to higher than the number of available cores, and don't take up all the cores on the machine.

Haplotype calling can be run using:

```
$ python /path/to/CallHap/CallHap_HapCallr.py \
  --inputHaps {RunID}_d250q20_Haps.vcf \
  --inputFreqs {RunID}_d250q20_Pools.vcf \
  -o {RunID} -p 20 -t 5 -l 2 --numRandom 100 –numTopRSS 3
```

`--inputHaps`	The haplotypes file from SNP filtering
`--inputFreqs`	The Pools file from SNP filtering
`-o`	A unique output prefix for this run of haplotype caller
`-p`	The number of individuals in each pool
`-t`	The number of threads to use during processing
`-l`	The number of times to iterate across the SNPs within each order
`-r`	How high a residual should be able to exist after adding a SNP, and is used to defer processing of a SNP where the residual doesn't reduce enough to another iteration.
`--dropFinal`	A flag which pairs with -r to remove SNPs with a high residual entirely at the end if they don't reduce the residual enough. May not work with current random ordering algorithm; don't use for now.
`--genpop`	A flag that instructs CallHap to generate genpop output
`--structure`	A flag that instructs CallHap to generate structure formated output
`--numRandom`	Controls how many psudo-random orderings of SNPs to use, and should be a value greater than zero. I set this value at 100 as a compromise between run time and increased chance of finding the correct solution; in practice, this value should be set based on the number of starting haplotypes relative to the number of SNPs present. If the number of starting haplotypes is close to the number of SNPs, this value can be low; the maximum number of haplotypes in the network is one more than the number of SNPs. However, if the number of SNPs is greater than the number of

	haplotypes, more attempts may be needed to help resolve the best network topology.
`--numTopRSS`	This option just influences how many RSS values down are processed for the final output solutions, and should be an integer greater than zero. I set it at a value of 3 so I could examine the higher RSS value solutions.

This program generates four to six output file per solution output (within the minimum number of RSS values):

- A NEXUS file (RunID_solNum_haps.nex) for network phylogeny creation; PopART (http://popart.otago.ac.nz/index.shtml) works fairly well. I've been using the TCS algorithm.
- A VCF file (RunID_solNum_PredFreqs.vcf) containing the estimated SNP frequencies based on the estimated haplotype frequencies, and the per-SNP average residuals in the INFO field
- A CSV file (RunID_solNum_freqs.csv) containing the per-pool haplotype frequencies and RSS values for each pool.
- A CSV file (RunID_solNum_Regression.csv) containing paired observed and predicted SNP frequencies from the Least Squared algorithm.
- (Optional): A Structure formatted file (RunID_solNum_iterNum.str) containing the expanded haplotype frequencies
- (Optional): A Genpop file containing the haplotype frequencies for use in Adigenet.

In addition, outputs are generated describing the original haplotypes network (RunID_Initial.nex), the unique haplotypes network (RunID_Unique.nex), raw topologies observed from each random order (RunID_RAW.csv), the frequency of each unique topology generated (RunID_topologies.csv), the frequency of occurrence for each haplotype found in any random order (RunID_summary.csv).

In terms of population-genetics analysis, haplotypes should be treated as independent alleles at a single locus.

Common Errors:

Problem	Solution
Quick-start pipeline produces empty files	Check that input files defined in the preconfig exist
Multiple best-RSS solutions	If one occurs more frequently than the other, use that one. If both occur equally, check to see if the network phylogenies for each solution look the same, and if the generated haplotype frequencies look the same. If the generated haplotype frequencies are identical, it doesn't matter which haplotype is actually present. If generated haplotype frequencies differ, create non-biologically relevant pools containing the same DNA samples, but shuffled in new ways (perhaps by using individual 1 from each population as one pool, individual 2 from each population as a second, and so on).

Appendix H: CallHap Programs

File structure

CallHap_VCF_Filt.py
CallHap_HapCallr.py
Modules
Modules/CallHap_LeastSquares.py
Modules/General.py
Modules/IO.py
Modules/VCF_parser.py
Modules/parallel.py

CallHap_VCF_Filt.py

```
#!/usr/bin/env python
# CallHap_VCF_Filt.py
# By Brendan F. Kohrn
# 3/20/2017
#
# This is the VCF filter used by the CallHap pipeline.

import numpy as np
from argparse import ArgumentParser
import time
from Modules.VCF_parser import *
from Modules.IO import *

parser = ArgumentParser()
parser.add_argument(
    "-i","--inVCF",
    action="store",
    dest="inFile",
    help="The input VCF file to be filtered.  All SSLs should be grouped \
          together in the first columns of the VCF, and all pools grouped \
          together afterwards, as in 'SSL1, SSL2, SSL3, ..., SSLN, Pool1, \
          Pool2, Pool3, ..., PoolM'.  ",
    required=True
    )
parser.add_argument(
    "-o", "--outHaps",
    action="store",
    dest="outHaps",
    help="A name for the output haplotypes VCF file." ,
    required=True
    )
parser.add_argument(
    "-O", "--outPools",
    action="store",
    dest="outPools",
    help="A name for the output pools VCF file.  ",
    required=True
    )
parser.add_argument(
    "-n", "--numSamps",
    action="store",
    dest="numSamps",
    type=int,
    help="The number of SSLs in your input VCF file",
    required=True
    )
parser.add_argument(
    "-N", "--numPools",
    action="store",
    dest="numPools",
    type=int,
    help="The number of pools in your input VCF file",
    required=True
    )
parser.add_argument(
    "-d", "--minDepth",
    action="store",
    dest="minDepth",
```

```
    type=int,
    help="The minimum depth to process a line, and the minimum average depth \
          to process a column.  ",
    default = 500
    )
parser.add_argument(
    "--minCallPrev",
    action="store",
    dest="minCallPrev",
    type=float,
    help="The percentage of reads that must be of a given identity in a SSL \
          to have that position be good.  ",
    default=0.9
    )
parser.add_argument(
    "--minSnpPrev",
    action="store",
    dest="minSnpPrev",
    type=float,
    help="The percent of a single individuals worth of reads that must be of a\
          given idetity to call a position as polymorphic based on pool \
          samples",
    default = 0.75
    )
parser.add_argument(
    "-p", "--poolSize",
    action="store",
    dest="poolSize",
    type=int,
    help="the number of individuals in each pooled library.  ",
    required=True
    )
parser.add_argument(
    "-q", "--minQual",
    action="store",
    dest="minQual",
    type=int,
    help="The minimum quality a given variant call must have to be processed.",
    default=100
    )
parser.add_argument(
    "--reportInterval",
    action="store",
    dest="rptInt",
    type=int,
    help="Report progress at this number of lines",
    default=1000
    )
parser.add_argument(
    "--dropLowDepth",
    action="store_true",
    dest="dropLow",
    help="Automatically drop any samples with an average depth under the \
          minimum depth.  "
    )
parser.add_argument(
    "--indelDist",
    action="store",
    dest="indelDist",
    default=100,
```

```
    help="How far away from indels to make SNPs.  Defaults to 100"
    )
o = parser.parse_args()

print("Running CallHap VCF filter on %s at %s" % (time.strftime("%d/%m/%Y"),
                                                  time.strftime("%H:%M:%S")))
pyCommand = "python CallHap_VCF_Filt.py --inVCF %s --outHaps %s " % (
    o.inFile, o.outHaps
    )
pyCommand += "--outPools %s --numSamps %s --numPools %s --minDepth %s " % (
    o.outPools, o.numSamps, o.numPools, o.minDepth
    )
pyCommand += "--minCallPrev %s --minSnpPrev %s --poolSize %s --minQual %s" % (
    o.minCallPrev, o.minSnpPrev, o.poolSize, o.minQual
    )

print("Command = %s" % pyCommand)
print("\nOpening files...")
# Open input VCF file
inVCF = vcfReader(o.inFile)
# Open output files
outHaps = open(o.outHaps, 'wb')
outPools = open(o.outPools, 'wb')
# Write command into header lines of both output files
outHaps.write("##Command=\"%s\"" % pyCommand)
outPools.write("##Command=\"%s\"" % pyCommand)

# Check average depth of each column in input
print("Checking depth of input columns...")
depths = [0. for x in xrange(o.numSamps + o.numPools)]
lines = 0
goodDepth = [True for x in xrange(o.numSamps + o.numPools)]
lineChekcer = []
goodVarCtr = 0
# Determine which columns have (on average) a good enough depth to pass the
# depth filter
for line in inVCF.lines:
    lineDPs = line.getData("DP","a")
    for iter1 in xrange(o.numSamps + o.numPools):
        if np.isnan(float(lineDPs[iter1])) == True:
            depths[iter1] += 0.
        else:
            depths[iter1] += float(lineDPs[iter1])
    lines += 1
for iter1 in xrange(o.numSamps + o.numPools):
    if depths[iter1]/lines >= o.minDepth:
        goodDepth[iter1] = True
    else:
        goodDepth[iter1] = False

vcfNames = inVCF.getNames()
# Print warnings about inadequate depth
if False in goodDepth:
    badColumns = [x for x in range(len(goodDepth)) if goodDepth[x] == False]
    for badIter in badColumns:
        print("\tWarning: Sample %s is has too low of a depth (%s)" %
              (vcfNames[badIter], depths[badIter]/lines))
        # If requested, automatically drop these columns
        if o.dropLow:
            print("; skipping\n")
```

```
        else:
            print("\n")
else:
    print("\tAll columns have greater than minimum average depth.  ")
# If dropping low depth columns has not been requested, reset goodDepth checker
if not o.dropLow:
    goodDepth = [True for x in xrange(o.numSamps + o.numPools)]

# Write column labels to both output files
outHaps.write(
    "\n#CHROM\tPOS\tID\tREF\tALT\tQUAL\tFILTER\tINFO\tFORMAT\t%s" % (
        "\t".join(
            [vcfNames[x] for x in  xrange(0,o.numSamps)
                if goodDepth[x] == True]
            ))
    )
outPools.write(
    "\n#CHROM\tPOS\tID\tREF\tALT\tQUAL\tFILTER\tINFO\tFORMAT\t%s" % (
        "\t".join([vcfNames[x] for x in  xrange(o.numSamps,
                                                o.numSamps+o.numPools)
                if goodDepth[x] == True ])
        )
    )

print("\nStarting analysis of lines...")
lineCounter = 0
indelLocs = []
outHapsLines = []
outPoolsLines = []
outLinesPoss = []
for line in inVCF.lines:
    # Print periodic progress reports
    if lineCounter % o.rptInt == 0 and lineCounter > 0:
        print("%s lines processed..." % lineCounter)
        print(line.getData("pos"))
        badReasons = [x[1] for x in lineChekcer]
        print("\nReport:")
        print("%s lines processed" % lineCounter)
        print("%s lines passing initial filters" %
              [x[0] for x in lineChekcer].count(True))
        print("\t%s good varients" % goodVarCtr)
        print("%s lines failing initial filters" %
              [x[0] for x in lineChekcer].count(False))
        print("\t%s incomplete coverage" %
            badReasons.count("incomplete coverage"))
        print("\t%s low depth" % badReasons.count("low depth"))
        print("\t%s too many differences between ref and one alt" %
            badReasons.count("too many differences between ref and one alt"))
        print("\t%s different differences between two alts and the ref" %
            badReasons.count("Different differences between two alts and the
ref"))
        print("\t%s unequal ref and alt lengths" %
            badReasons.count("unequal ref and alt lengths"))
        print("\t%s alt longer than 1 with ref length of 1" %
            badReasons.count("alt longer than 1 with ref length of 1"))
        print("\t%s Low quality variant call" %
            badReasons.count("low quality SNP call"))

    lineCounter += 1
    # Retrieve basic information about this line
```

```
pos = line.getData("pos")
lineRefCounts = line.getData("RO","a")
lineDPs = line.getData("DP","a")
useLine = True
# Check that there is depth in all samples for this line
if np.nan in lineDPs:
    lineChekcer.append((False, "incomplete coverage", pos))
    useLine = False
# Check that this line passes the quality filter
elif float(line.getData("qual")) < o.minQual:
    lineChekcer.append((False, "low quality SNP call", pos))
    useLine = False
# Check that all used columns in this line have adequate depth
elif False in [
    True if int(lineDPs[x]) >= o.minDepth or goodDepth[x] == False
    else False for x in xrange(len(lineDPs))
    ]:
    lineChekcer.append((False, "low depth", pos))
    useLine = False
# Check that the length of the reference is 1
elif len(line.getData("ref")) > 1:
    # If the length of the reference is greater than 1, check that the
    # length of the alt matches the length of the reference
    # And that all alt alleles are the same length
    if (
     max([len(x) for x in line.getData("alt")]) == len(line.getData("ref"))
     and len(set([len(y) for y in line.getData("alt")])) <= 1
     ):
        altValues = line.getData("alt")
        refValue = line.getData("ref")
        newAlt = []
        diffIdxs = []
        # If ref and alt alleles are the same length, check that there is
        # only one difference between them
        for altValue in altValues:
            numDiffs = 0
            diffIdx = []
            for diffCounter in xrange(len(altValue)):
                if refValue[diffCounter] != altValue[diffCounter]:
                    numDiffs += 1
                    diffIdx.append(diffCounter)
            # If there is more than one difference between ref and alt
            # alleles, discard the line
            if numDiffs > 1:
                if useLine == True:
                    lineChekcer.append(
                        (False,
                         "too many differences between ref and one alt",
                         pos)
                        )
                useLine = False

            else:
                # Calculate which base pairs are different between this alt
                # and the reference
                diffIdxs.append(diffIdx[0])
                newAlt.append(altValue[diffIdx[0]])
        # Check that all alts have the same base pair different from the
        # reference
        if len(set(diffIdxs)) == 1:
```

```
            # If they do, change the ref and alt alleles, and the position
            # accordingly
            line.setElmt("pos", line.getData("pos") + diffIdx[0])
            line.setElmt("ref", refValue[diffIdxs[0]])
            line.setElmt("alt", newAlt)
        # Otherwise, discard the line
        else:
            useLine = False
            lineChekcer.append(
                (False,
                "Different differences between two alts and the ref",
                pos)
                )
            # Keep track of this location as the location of an indel
            indelLocs.append(int(line.getData("pos")))
    else:
        # If ref and alt are different lengths, discard the line
        useLine = False
        lineChekcer.append((False, "unequal ref and alt lengths", pos))
        indelLocs.append(int(line.getData("pos")))
# Check that the length of the alt allele is 1
elif max([len(x) for x in line.getData("alt")]) > 1:
    # If not, discard the line
    lineChekcer.append(
    (False, "alt longer than 1 with ref length of 1", pos)
    )
    useLine = False
    # Keep track of this location as the location of an indel
    indelLocs.append(int(line.getData("pos")))
if useLine == True:
    # If this line has not been discarded yet,
    lineChekcer.append([True, "good line", pos])
    # Count alternate alleles for the line
    lineAltCounts = [[x] for x in line.getData("AO", "a")]
    sampIDs = []
    conflicted = False
    monomorphic = False
    # Determine the identity (Ref/Alt) of each sample (SSL) in this line
    for sampIter in xrange(o.numSamps):
        if goodDepth[sampIter] == True:
            sampTest = [int(lineRefCounts[sampIter])]
            sampTest.extend([int(x) for x in lineAltCounts[sampIter]])
            maxIter = None
            for iter1 in xrange(len(sampTest)):
                if maxIter == None:
                    maxIter = iter1
                elif sampTest[iter1] > sampTest[maxIter]:
                    maxIter = iter1
            sampIDs.append(maxIter)
            # Test if there are no reads in any of the identities for this
            # sample/line
            if sum(sampTest) == 0:
                if conflicted == False:
                    lineChekcer[-1].append(
                        "conflicted because sum sampTest = 0 (line 167)"
                        )
                conflicted = True
            # Test if the proportion of the most common identity in this
            # sample/line is high enough to reliably call
            elif float(sampTest[maxIter])/sum(sampTest) < o.minCallPrev:
```

```
            if conflicted == False:

                lineChekcer[-1].append(
                    "conflicted because of minCallPrev (line 170)"
                    )
                conflicted = True
# If no conflicts exist
if conflicted == False:
    # Figure out the reference allele
    refAllele = None
    altAllele = None
    for iter1 in [0, 1, 2]:
        if refAllele == None:
            refAllele = iter1
        elif sampIDs.count(iter1) > sampIDs.count(refAllele):
            altAllele = refAllele
            refAllele = iter1
        elif altAllele == None:
            altAllele = iter1
        elif sampIDs.count(iter1) > sampIDs.count(altAllele):
            altAllele = iter1
    # Calculate ref allele frequency in each pool
    poolFreqs = []
    for iter1 in xrange(o.numPools):
        if goodDepth[o.numSamps + iter1] == True:
            if refAllele == 0:
                poolFreqs.append(
                    float(lineRefCounts[o.numSamps + iter1])/
                    int(lineDPs[o.numSamps + iter1])
                    )
            else:
                poolFreqs.append(float(
                    lineAltCounts[o.numSamps + iter1][refAllele - 1])/
                    int(lineDPs[o.numSamps + iter1]))
    monomorphicSamps = False
    polymorphicPools = False
    # Check if locus is monomorphic in single samples
    if sampIDs.count(refAllele) == len(sampIDs):
        monomorphicSamps = True
        lineChekcer[-1].append("monomorphicSamps")
    # Check if sample is polymorphic in pools
    if o.numPools > 0:
        if min(poolFreqs) <= 1. - (o.minSnpPrev/o.poolSize):
            polymorphicPools = True
            lineChekcer[-1].append("polymorphicPools")
    # If either SSLs are polymorphic or SSLs are monomorphic and Pools
    # are polymorphic, keep line
    if monomorphicSamps==False or (polymorphicPools == True and
                                   monomorphicSamps == True):
        goodVarCtr += 1
        lineRef = line.getData(
            "ref" if refAllele == 0 else "alt")[0 if refAllele == 0
                                                else refAllele - 1]
        lineAlt = line.getData(
            "ref" if altAllele == 0 else "alt")[0 if altAllele == 0
                                                else altAllele - 1]
        linePos = line.getData("pos")
        lineChrom = line.getData("chrom")
        lineQual = line.getData("qual")
```

```
                outHapsLines.append(
                    "\n%s\t%s\t.\t%s\t%s\t%s\t.\t.\tGT\t%s" % (lineChrom,
                        linePos,
                        lineRef,
                        lineAlt,
                        lineQual,
                        "\t".join(["0" if x == refAllele else "1"
                                   for x in sampIDs]))
                    )
                outPoolsLines.append(
                    "\n%s\t%s\t.\t%s\t%s\t%s\t.\t.\tRF\t%s" % (lineChrom,
                        linePos,
                        lineRef,
                        lineAlt,
                        lineQual,
                        "\t".join([str(x) for x in poolFreqs]))
                    )
                outLinesPoss.append(int(line.getData("pos")))
# Check all variants located so far for proximity to indels
finGoodVars = 0
for outIter in xrange(len(outLinesPoss)):
    currPos = outLinesPoss[outIter]
    useVar = True
    indelIter = 0
    indelPos = 0
    while indelPos <= currPos + o.indelDist and indelIter < len(indelLocs):
        indelPos = indelLocs[indelIter]
        if indelPos > currPos and indelPos - 10 < currPos:
            useVar = False
        elif indelPos < currPos and indelPos + 10 > currPos:
            useVar = False
        elif indelPos == currPos:
            useVar = False
        indelIter += 1
    # Write output files
    if useVar == True:
        outHaps.write(outHapsLines[outIter])
        outPools.write(outPoolsLines[outIter])
        finGoodVars += 1

outHaps.close()
outPools.close()

# Create Nexus output
finOutToNex,finOutNames = toNP_array(o.outHaps,"GT")
if finOutToNex.shape[0] != 0:
    NexusWriter(
        [vcfNames[x] for x in  xrange(0,o.numSamps) if goodDepth[x] == True],
        finOutToNex,
        finOutToNex.shape[0],
        o.outHaps[:-4],
        "",
        o.outHaps
        )
#output report
badReasons = [x[1] for x in lineChekcer]
print("\nFinal report:")
print("End time = %s %s" %
    (time.strftime("%d/%m/%Y"),time.strftime("%H:%M:%S")))
print("%s lines processed" % lineCounter)
```

```
print("%s lines passing initial filters" %
    [x[0] for x in lineChekcer].count(True))
print("\t%s good variants" % goodVarCtr)
print("\t%s not within %s bp of an indel" % (finGoodVars,o.indelDist))
print("%s lines failing initial filters" %
    [x[0] for x in lineChekcer].count(False))
print("\t%s incomplete coverage" % badReasons.count("incomplete coverage"))
print("\t%s low depth" % badReasons.count("low depth"))
print("\t%s too many differences between ref and one alt" %
    badReasons.count("too many differences between ref and one alt"))
print("\t%s different differences between two alts and the ref" %
    badReasons.count("Different differences between two alts and the ref"))
print("\t%s unequal ref and alt lengths" %
    badReasons.count("unequal ref and alt lengths"))
print("\t%s alt longer than 1 with ref length of 1" %
    badReasons.count("alt longer than 1 with ref length of 1"))
print("\t%s Low quality variant call" %
    badReasons.count("low quality SNP call"))
```

CallHap_HapCallr.py

```
#!/bin/python
# CallHap_HapCallr V. 1.01.00
#
# A program for determining full-genome haplotype frequencies in pooled DNA
# samples based on SNP calls and limited known haplotypes.
# Takes as input a pair of VCF files describing haplotype identity and SNP
# frequency, as generated by CallHap VCF_Filt
#
# Import necessary modules
import numpy as np
from argparse import ArgumentParser
import time
import sys
import random
from multiprocessing import Pool
from Modules.VCF_parser import *
from Modules.CorrHaps import *
from Modules.CallHap_LeastSquares import *
from Modules.General import *
from Modules.IO import *
from Modules.parallel import *

progVersion = "V1.01.00"

def  MakeHaps(inSnpSets, inPoolSize, inOldHaps, inInitialFreqs, InitialHaps):
    # Module to create new haplotypes using input SNP sets and haplotype set.
    # Figure out what the less common identity for this SNP is in the current
    # haplotype set
    snpIDs = [inOldHaps[x][inSnpSets[0]] for x in xrange(len(inOldHaps))]
    numSnps = len(inOldHaps[0])
    commonCounter = [snpIDs.count(0), snpIDs.count(1)]
    if commonCounter[0] > commonCounter[1]:
        rareAllele=1
    else:
        rareAllele=0
    # Figure out which haplotypes contain the less common variant
    containingHaps = [True if inOldHaps[x][inSnpSets[0]] == rareAllele
                      else False for x in xrange(len(inOldHaps))]
    if True in containingHaps: # If this SNP is in a known haplotype
        # Determine which SNPs can be legally changed in each haplotype
        legalSnpsByHap = ValidSnpsFromPhylogeny(inOldHaps)
        # Check which haplotypes the target SNP can be legally changed in
        # These are the ones that could be used to create new source haplotypes
        usableHaps = [True if inSnpSets[0] in legalSnpsByHap[hap] else False
                      for hap in xrange(len(inOldHaps))]
    else: # If this SNP is not in a known haplotype
        # All haplotypes can be used to create new source haplotypes.
        usableHaps = [True for x in containingHaps]
    # Initialize lists of possible haplotype sets
    possibleFreqs = [inInitialFreqs[:]]
    possibleHaps = [inOldHaps]
    initialHaps = len(inOldHaps)

    freqSet = 0
    testStop = len(possibleFreqs)
    loopCtr1 = 0

    # while there are still haplotypes to try adding this SNP to
```

```
    while freqSet < testStop:
        loopCtr1 += 1
        baseFreq = []
        for freq in xrange(len(possibleFreqs[freqSet])):

            if possibleFreqs[freqSet][freq] > 0 and usableHaps[freq] == True:
                baseFreq.append(freq)

        newFreq = 0
        loopCtr2 = 0
        while newFreq < len(baseFreq):
            loopCtr2 += 1
            if loopCtr2 > 1000:
                raise Exception(
                    "Too many iterations at line 342 with baseFreq = %s" %
                    len(baseFreq)
                    )
            if baseFreq[newFreq] > initialHaps:
                if newFreq == len(baseFreq) - 1:
                    # Change the original frequency set and haplotypes set
                    possibleFreqs[freqSet].append(1)
                    possibleHaps[freqSet].append(
                        np.copy(possibleHaps[freqSet][baseFreq[newFreq]])
                        )
                    for iter1 in inSnpSets:
                        possibleHaps[freqSet][-1][iter1] = 1 -
possibleHaps[freqSet][-1][iter1]
                else:
                    # make a copy of the original frequency set and haplotypes
                    # set
                    possibleFreqs.append([x for x in possibleFreqs[freqSet]])
                    possibleHaps.append([np.copy(x)
                        for x in possibleHaps[freqSet]])
                    # change the copy
                    possibleFreqs[-1].append(1)
                    possibleHaps[-1].append(
                        np.copy(possibleHaps[freqSet][newFreq])
                        )
                    for iter1 in inSnpSets:
                        possibleHaps[-1][-1][iter1] = 1 - possibleHaps[-1][-
1][iter1]
            else:

                if newFreq == len(baseFreq) - 1:
                    # Change the original frequency set and haplotypes set
                    possibleFreqs[freqSet].append(1)
                    possibleHaps[freqSet].append(
                        np.copy(possibleHaps[freqSet][baseFreq[newFreq]])
                        )
                    for iter1 in inSnpSets:
                        possibleHaps[freqSet][-1][iter1] = 1 -
possibleHaps[freqSet][-1][iter1]

                    if possibleFreqs[freqSet][baseFreq[newFreq]] == 0:
                        possibleFreqs[freqSet].pop(baseFreq[newFreq])
                        possibleHaps[freqSet].pop(baseFreq[newFreq])
                else:
                    # make a copy of the original frequency set and haplotypes
                    # set
                    possibleFreqs.append([x for x in possibleFreqs[freqSet]])
```

```
                    possibleHaps.append([np.copy(x)
                                         for x in possibleHaps[freqSet]])
                    # change the copy
                    possibleFreqs[-1].append(1)
                    possibleHaps[-1].append(
                        np.copy(possibleHaps[freqSet][baseFreq[newFreq]])
                        )
                    for iter1 in inSnpSets:
                        possibleHaps[-1][-1][iter1] = 1 - possibleHaps[-1][-
1][iter1]
                    if (possibleFreqs[freqSet][baseFreq[newFreq]] == 0
                        and baseFreq[newFreq] >= InitialHaps):
                        possibleFreqs[freqSet].pop(baseFreq[newFreq])
                        possibleHaps[freqSet].pop(baseFreq[newFreq])
            newFreq += 1
        freqSet += 1
    return(possibleHaps)

def CallHapMain(OrderNumber, o):
    print("Starting Random Order %s/%s" % (str(OrderNumber + 1),
                                           str(o.numRand)))
    # Load haplotypes
    KnownHaps, KnownNames = toNP_array(o.knownHaps, "GT")
    # Invert haplotypes so that ref allele is 1
    KnownHaps = invertArray(KnownHaps)
    # Find unique haplotypes
    inHapArray, UniqueNames = UniqueHaps(KnownHaps, KnownNames)
    # Count number of unique haplotypes
    numHapsInitial = len(UniqueNames)
    # Count number of SNPs
    numSNPs = inHapArray.shape[0]
    # Add "dummy" SNP to ensure haplotype frequencies sum correctly
    inHapArray = ExtendHaps(inHapArray)
    # Store input haplotypes in bestArray
    bestArray = np.copy(inHapArray)
    # Load SNPs
    SnpFreqs, poolNames = toNP_array(o.inFreqs, "RF")
    # Add "dummy" SNP to ensure haplotype frequencies sum correctly
    SnpFreqs = ExtendHaps(SnpFreqs)
    # Count number of pools present
    numPools = len(poolNames)

    # Count number of haplotypes again to save initial number of known
    # haplotypes for later
    # May not be needed in random method
    numHapsInitial1 = len(UniqueNames)
    # Set baseNumHapSets to keep track of source haplotype set for each created
    # haplotype set
    baseNumHapSets = 1
    # Convert haplotypes and SNPs arrays to decimal format to prevent rounding
    # errors
    bestArray = npToDecNp(bestArray)
    SnpFreqs = npToDecNp(SnpFreqs)

    # Find base SLSq
    # Save base RSS
    baseSLSq = []
    # Save base haplotype frequencies
    baseFreqs = []
    # save base residuals
```

```
baseResiduals = [[]]
# Calculate RSS for each pool
for poolIter in xrange(numPools):
    tmpSol = Find_Freqs(bestArray, SnpFreqs[:,poolIter], o.poolSize)
    baseSLSq.append(tmpSol[1])
    baseFreqs.append(tmpSol[0])
    baseResiduals[0].append(
        np.array([[x] for x in list(residuals(tmpSol[0][0],bestArray,
                    SnpFreqs[:,poolIter],o.poolSize))])
        )
# Calculate total per SNP RSS values for all SNPs; method for deterministic
# ordering
baseSnpResids = [sum([baseResiduals[0][pool][xSnp]
    for pool in xrange(numPools)]) for xSnp in xrange(numSNPs)]
# Find overall SNP frequency in SSLs; method for deterministic ordering
snpFreqsTotal = np.sum(bestArray, axis=1) < bestArray.shape[1]

# Create random SNP ordering
snpCombins3 = [[x] for x in range(numSNPs)]
random.shuffle(snpCombins3)
snpCombins3 = [y for y in sorted(snpCombins3,
    key = lambda x: snpFreqsTotal[x[0]], reverse = True)]
print("SNP Order %s/%s: \n%s" % (str(OrderNumber + 1),
                                 str(o.numRand), snpCombins3))

#Find base average RSS value
baseRSS = sum(baseSLSq)/len(baseSLSq)

fullFreqs = [[0 for x in xrange(numHapsInitial)]]
for testIter in xrange(numHapsInitial):
    for testIter2 in xrange(numPools):
        if baseFreqs[testIter2][0,testIter] > 0:
            fullFreqs[0][testIter] = 1
# Break up haplotypes array into a list of arrays
potHapSets = [[np.copy(bestArray[:,x]) for x in xrange(numHapsInitial)]]
numHaps = [numHapsInitial]
# AIC and RSS are used somewhat interchangeably as variable names
# in this program at the moment, and I don't have the time to clean it up
# right now.  It will be cleaned up in the future
bestAIC = [baseRSS]
usedSnps = 0
# Start adding SNPs
# In the case of multiple iterations:
for iteration in xrange(o.numIterations):
    # Legacy line from when I was grouping SNPs based on correlation,
    # or residual, or frequency
    for combin in snpCombins3:
        # Keep track of where in the list of SNPs I am so the user knows
        # something's happening
        usedSnps += 1
        # Test if this SNP combination has any non-zero residuals
        useCombin = False
        for hapSetIter in xrange(baseNumHapSets):
            for population in xrange(numPools):
                if abs(round(
                        20*baseResiduals[hapSetIter][population][combin[0]]
                        )) > 0:
                    useCombin = True
        # If this SNP combination has non-zero residuals:
        if useCombin:
```

```
newPotHapSets = []
potHapSetsAIC = []
sourceHapSet = []
newFullFreqs = []
# Find options for adding this SNP set:
currentHapSet = 0
snpRes = []
for hapSet in potHapSets:
    newPotHaps = MakeHaps(combin, o.poolSize, copy(hapSet),
                          fullFreqs[0], numHapsInitial)
    SLSqs = []
    Freqs = []
    testAICList = []
    maxRSSList = []
    srcHap = []
    # Find the average SLSq for each pot hap set
    newPotHaps2 = []
    intermediate = []
    for solverIter1 in xrange(len(newPotHaps)):
        intermediate.append(
            easyConcat(newPotHaps[solverIter1])
            )
    cleanedIntermediate = [x for x in intermediate
                           if not x is None]
    func = partial(massFindFreqs, inSnpFreqs=SnpFreqs,
                   p=o.poolSize)
    result = []
    for solverIter in xrange(len(cleanedIntermediate)):
        result.append(func(cleanedIntermediate[solverIter]))
    tmpSols = [x for x in result if not x is None]

    # Determine which solutions (and thus haplotypes) produce
    # an improvement in RSS value
    testAICList = [x for x in xrange(len(tmpSols))
                   if tmpSols[x][2] <= bestAIC[currentHapSet]]
    # Keep track of the source haplotye set for these solutions
    srcHap = [currentHapSet for x in xrange(len(testAICList))]
    # Calculate per SNP residuals to test if improvement was
    # enough to keep this SNPs solutions
    newResiduals = []
    changedResids = []
    SnpResiduals = []
    solIter = 0
    for sol in tmpSols:
        newFullFreqs.append(
            [0 for x in xrange(len(sol[1][0][0]))]
            )
        for testIter in xrange(len(newFullFreqs[-1])):
            for testIter2 in xrange(numPools):
                if sol[1][testIter2][0,testIter] > 0:
                    newFullFreqs[-1][testIter] = 1
                newResiduals.append(
                  np.array([[x]
                  for x in list(residuals(sol[1][testIter2][0],
                  np.concatenate([np.transpose(y[np.newaxis])
                  for y in newPotHaps[solIter]], axis=1),
                  SnpFreqs[:,testIter2],o.poolSize))])
                  )
        # Calculate per SNP RSS values
        SnpResiduals.append(
```

```
                [sum([newResiduals[poolIter][x]**2
                for poolIter in xrange(numPools)])/numPools
                for x in xrange(numSNPs)]
                )
            solIter += 1
        snpRes1 = [SnpResiduals[x][combin[0]]
                   for x in xrange(len(tmpSols))]
        if len(testAICList) > 0:
            # Filter to only the best solutions out of all proposed
            # solutions based on this haplotype set
            # Sort solutions better than starting RSS by RSS value,
            # from lowest to highest
            testIndex = sorted(testAICList,
                               key=lambda x: tmpSols[x][2])
            # If no best solution for this SNP exists, the best
            # solution for this
            if len(potHapSetsAIC) == 0:
                testFreq = tmpSols[testIndex[0]][2]
            # If the best RSS from this solution is worse than the
            # best RSS so far proposed, use the best RSS so far
            # proposed
            elif tmpSols[testIndex[0]][2] >= min(potHapSetsAIC):
                testFreq = min(potHapSetsAIC)
            # Otherwise, use the best RSS value from this SNP
            else:
                testFreq = tmpSols[testIndex[0]][2]
            # If this RSS value represents an improvement, sort and
            # save solutions
            if testFreq < bestAIC[currentHapSet]:
                iter1 = 0
                minAICIndex = []
                continueLoop = True
                # Save all solutions (and thus potential haplotype
                # sets) that represent an improvement in RSS value
                while (iter1 < len(testIndex) and
                       tmpSols[testIndex[iter1]][2] <= testFreq):
                    newPotHapSets.append(
                        copy(newPotHaps[testIndex[iter1]])
                        )
                    potHapSetsAIC.append(
                        tmpSols[testIndex[iter1]][2]
                        )
                    sourceHapSet.append(currentHapSet)
                    snpRes.append(snpRes1[testIndex[iter1]])
                    iter1 += 1
            else:
                minAICIndex = []
        # Next haplotype set
        currentHapSet += 1
    # Check if the ending residual values for a SNP are too high
    continueCheck = [False if snpRes[x] >= o.highResidual else True
                     for x in xrange(len(snpRes))]
    # Sort potential haplotype sets by RSS value
    bestAICIdx = sorted(range(len(newPotHapSets)),
                        key=lambda x: potHapSetsAIC[x])
    # Filter solutions based on RSS values, keeping only the lowest
    # RSS values
    if len(bestAICIdx) > 0 and True in continueCheck:
        bestFreq = potHapSetsAIC[bestAICIdx[0]]
        potHapSets = []
```

```
            bestAIC = []
            iter1 = 0
            minCtr = 0
            newSourceHap = []
            potHapSetsMaxRSS = []
            while  iter1 < len(bestAICIdx):
                if (potHapSetsAIC[bestAICIdx[iter1]] == bestFreq and
                        snpRes[bestAICIdx[iter1]] < o.highResidual):
                    minCtr += 1
                    potHapSets.append(
                        copy(newPotHapSets[bestAICIdx[iter1]])
                        )
                    bestAIC.append(potHapSetsAIC[bestAICIdx[iter1]])
                    newSourceHap.append(
                        sourceHapSet[bestAICIdx[iter1]]
                        )
                iter1 += 1
            fullFreqs = newFullFreqs[:]
            sourceHapSet = newSourceHap[:]
            bestRSS = bestFreq
            numHaps = [len(x) for x in potHapSets]
# Filter any solutions that made it through all SNPs. to only those
# with the lowest AIC (this time, really is AIC value)
SLSqs = []
Freqs = []
finFullFreqs = []
SolutionHapSets = []
SolutionAICs = []
# Remove unused haplotypes from each potential final hap set
intermediate = []
for solverIter1 in xrange(len(potHapSets)):
    intermediate.append(easyConcat(potHapSets[solverIter1]))
cleanedIntermediate = [x for x in intermediate if not x is None]
func = partial(massFindFreqs, inSnpFreqs=SnpFreqs, p=o.poolSize)
result = []
for solverIter in xrange(len(cleanedIntermediate)):
    result.append(func(cleanedIntermediate[solverIter]))
tmpSols = [x for x in result if not x is None]

for sol in tmpSols:
    finFullFreqs.append([0 for x in xrange(len(sol[1][0][0]))])
    for testIter in xrange(len(finFullFreqs[-1])):
        for testIter2 in xrange(numPools):
            if sol[1][testIter2][0,testIter] > 0:
                finFullFreqs[-1][testIter] = 1
# Calculate AIC values for each solution
SolutionAICs = [AIC_from_RSS(tmpSols[x][2],
                             sum(finFullFreqs[x]), numSNPs)
                for x in xrange(len(tmpSols))]
# Create solution haplotype sets with only haplotypes present in
# initial haplotypes or with frequency in pools
# Known haplotypes should be a subset of haplotypes with frequency in
# the final solution, but this is just in case they aren't
SolutionHapSets = [[np.copy(potHapSets[x][y])
                   for y in xrange(len(finFullFreqs[x]))
                   if finFullFreqs[x][y] > 0 or y < numHapsInitial ]
                   for x in xrange(len(tmpSols))]
# If SNPs are being removed permenantly after the final iteration:
if o.dropFinal == True and iteration == o.numIterations - 1:
    # Figure out which SNPs to remove for each proposed solution
```

```
    newResiduals = []
    snpsToRemove = []
    solIter = 0
    for sol in tmpSols:
        for testIter in xrange(len(newFullFreqs)):
            for testIter2 in xrange(numPools):
                newResiduals.append(
                    np.array([[x] for x in list(
                        residuals(sol[1][testIter2][0],
                           np.concatenate([np.transpose(y[np.newaxis])
                               for y in potHapSets[solIter]], axis=1),
                           SnpFreqs[:,testIter2],o.poolSize)
                       )])
                   )
        SnpResiduals = [sum([newResiduals[poolIter][x]**2
                        for poolIter in xrange(numPools)])
                        for x in xrange(numSNPs)]
        snpsToRemove.append([])
        for snpRemovalIter in xrange(numSNPs):
            if SnpResiduals[snpRemovalIter] >= o.highResidual:
                snpsToRemove[-1].append(snpRemovalIter)
        solIter += 1
else:
    snpsToRemove = [[] for x in xrange(len(tmpSols))]



# Figure out which solution(s) has (have) the lowest AIC
AIC_test_idx = sorted(range(len(SolutionAICs)),
                      key = lambda x: SolutionAICs[x])
finIndex = 0
testFreq = SolutionAICs[AIC_test_idx[0]]
iter1 = 0
minAICIndex = []
continueLoop = True
# Figure out how many solutions to output
while  iter1 < len(AIC_test_idx) and continueLoop:
    if SolutionAICs[AIC_test_idx[iter1]] == testFreq:
        finIndex += 1
    else:
        continueLoop = False
    iter1 += 1

# Start resetting base haplotype residuals
baseResiduals = []

# Output solutions
newPotHapSets = []
bestAIC = []
if iteration == o.numIterations - 1:
    outputList = []
for outputIdx in xrange(finIndex):
    if iteration == o.numIterations - 1:
        outputList.append([])
    # Create final haplotypes array
    finSolution = np.concatenate(
        [SolutionHapSets[
            AIC_test_idx[outputIdx]][x][np.newaxis].transpose()
            for x in xrange(len(
```

```
            SolutionHapSets[AIC_test_idx[outputIdx]]
            ))]
    , axis=1
    )
# Remove any SNPs that need removing
finSolution = np.delete(finSolution, snpsToRemove[outputIdx], 0)
# Find (or make) haplotype names
myHapNames = []
newHapNumber = 1
for haplotypeIter in xrange(finSolution.shape[1]):
    if haplotypeIter >= len(UniqueNames):
        # For new haplotypes, build a new haplotype name, keeping
        # track of iteration and new haplotype number
        myHapNames.append(
            "NewHap_%s.%s" % (str(iteration).zfill(2),
                              str(newHapNumber).zfill(2)))
        newHapNumber += 1
    else:
        # For known haplotypes, use the original haplotype name
        myHapNames.append(UniqueNames[haplotypeIter])
# Redo uniqueness of haplotypes in case removing a SNP merged two
# haplotypes
finSolution, finNames = UniqueHaps(finSolution, myHapNames)
# remove SNPs from SNP frequencies
finSNPs = np.delete(SnpFreqs,snpsToRemove[outputIdx],0)

# Create decimal haplotype identifiers
myDecHaps = []
for haplotypeIter in xrange(finSolution.shape[1]):
    myDecHaps.append(int("1"+"".join([str(int(x))
        for x in finSolution[:, haplotypeIter]]),2))
if iteration == o.numIterations - 1:
    outputList[-1].append(myDecHaps)

SLSqs = []
Freqs = []
predSnpFreqs = []
newResiduals = []

for poolIter in xrange(numPools):
    tmpSol = Find_Freqs(finSolution, finSNPs[:,poolIter],
                        o.poolSize)
    SLSqs.append(tmpSol[1])
    Freqs.append(tmpSol[0])
    # Calculate residuals for this pool
    newResiduals.append(
        np.array([[x] for x in list(residuals(tmpSol[0][0],
                                              finSolution,
                                              finSNPs[:,poolIter],
                                              o.poolSize))])
        )
    # Calculate predicted SNP frequencies
    predSnpFreqs = np.sum(finSolution * tmpSol[0][0],
                          axis = 1)/o.poolSize
if iteration == o.numIterations - 1:
    outputList[-1].append(average(SLSqs))
baseResiduals.append(newResiduals[:])
# Calculate per SNP RSS values for VCF output
SnpResiduals = [float(sum([newResiduals[poolIter][x]**2
    for poolIter in xrange(numPools)])[0])
```

```
                for x in xrange(numSNPs-len(snpsToRemove[outputIdx]))]
            bestAIC.append(sum(SLSqs)/len(SLSqs))
            # Save this haplotype set for the next iteration
            newPotHapSets.append([np.copy(finSolution[:,x])
                for x in xrange(finSolution.shape[1])])
        # Setup for next iteration
        usedSnps = 0
        numHapsInitial = len(myHapNames) # may need some fixing
        UniqueNames = myHapNames[:] # may need some fixing
        numHaps = [numHapsInitial for x in xrange(len(newPotHapSets))]
        outPrefix = "%s_Iteration%s" % (o.outPrefix, iteration + 2)
        potHapSets = newPotHapSets[:]
        fullFreqs = [[1 for x in xrange(len(potHapSets[y]))]
            for y in xrange(len(potHapSets))]
        # Go on to the next iteration
        if iteration == o.numIterations - 1:
            print("Finished Random Order %s/%s" % (str(OrderNumber + 1),
                                                   str(o.numRand)))
            return(outputList)

if __name__ == "__main__":
    # Load options
    parser = ArgumentParser()
    parser.add_argument(
        '-i','--inputHaps',
        action="store",
        dest="knownHaps",
        help = "A VCF-formatted file containing the known haplotypes encoded \
                in the GT field.  GT must be present in the FORMAT field, and \
                ploidy must be 1.  ",
        required=True
        )
    parser.add_argument(
        '-p', '--poolsize',
        action="store",
        type=int,
        dest="poolSize",
        help="The number of individuals in each pool.  ",
        required=True
        )
    parser.add_argument(
        '-f','--inputFreqs',
        action="store",
        dest="inFreqs",
        help="A VCF-formatted file containing the input pool frequencies \
              encoded in the RF field.  RF must be present in the FORMAT \
              field.  ",
        required=True
        )
    parser.add_argument(
        '-o','--outPrefix',
        action="store",
        dest="outPrefix",
        required=True,
        help="A prefix for output file names.  "
        )
    parser.add_argument(
        "-v", "--version",
        action="store_true",
        dest="v",
```

```
    help="Displays the version number and exits."
    )
parser.add_argument(
    '-t', '--processes',
    type=int,
    action="store",
    dest="numProcesses",
    default=None,
    help="The number of processes to use.  Should not be more than the \
          number of cores on your CPU.  Defaults to using the number of \
          cores on your CPU.  "
    )
parser.add_argument(
    '-l','--numIterations',
    type=int,
    action="store",
    dest="numIterations",
    default=1,
    help="Number of iterations to run within each random ordering."
    )
parser.add_argument(
    '-r','--highResidual',
    type=float,
    action="store",
    dest="highResidual",
    default=100,
    help="Cutoff value for delaying processing of a SNP until after all \
          other SNPs have been processed"
    )
parser.add_argument(
    '--dropFinal',
    action="store_true",
    dest="dropFinal",
    help="If after delaying processing on a SNP, the solution isn't \
          improved by keeping it, drop the SNP.  If absent, the SNP will \
          be processed as normal at the end.  "
    )
parser.add_argument(
    '--genpop',
    action="store_true",
    dest="genpopOutput",
    help="Output a genpop file of the resulting haplotype frequencies.  "
    )
parser.add_argument(
    '--structure',
    action="store_true",
    dest="strOutput",
    help="Output a Structure formatted file of the resulting haplotype \
          frequencies.  "
    )
parser.add_argument(
    '--numRandom',
    type=int,
    action="store",
    dest="numRand",
    help="The number of random orders to use for haplotype creation.  \
          More orders will yield more accurate results, but will also \
          take longer.  "
    )
parser.add_argument(
```

```
    '--numTopRSS',
    type=int,
    action="store",
    dest="topNum",
    default=3,
    help="The number of top RSS values you want to output files for.  \
         Increasing the size of this number may lead to a large number of \
         outputs.  "
    )
o = parser.parse_args()

# version output
if o.v:
    print(progVersion)
    exit()

# Print initialization text
print("Running CallHap on %s at %s:" % (time.strftime("%d/%m/%Y"),
                                        time.strftime("%H:%M:%S")))
CommandStr = "Command = python CallHap_HapCallr.py"
CommandStr += "--inputHaps %s " % o.knownHaps
CommandStr += "--inputFreqs %s " % o.inFreqs
CommandStr += "--poolSize %s " % o.poolSize
CommandStr += "--outPrefix %s " % o.outPrefix
CommandStr += "--processes %s " % o.numProcesses
CommandStr += "--numIterations %s " % o.numIterations
CommandStr += "--highResidual %s " % o.highResidual
if o.dropFinal:
    CommandStr += "--dropFinal "
if o.genpopOutput:
    CommandStr += "--genpop "
if o.strOutput:
    CommandStr += "--structure "
CommandStr += "--numRandom %s " % o.numRand
CommandStr += "--numTopRSS %s" % o.topNum
print(CommandStr)

# Set initial output prefix
outPrefix = "%s" % (o.outPrefix)

pool = Pool(processes=o.numProcesses, maxtasksperchild=500)
func = partial(CallHapMain, o=o)
funcIterable = range(o.numRand)
result = pool.map(func, funcIterable)
cleaned = [x for x in result if not x is None]
# not optimal but safe
pool.close()
pool.join()

## Get initial haplotypes / SNP frequencies
# Load haplotypes
KnownHaps, KnownNames = toNP_array(o.knownHaps, "GT")
# Invert haplotypes so that ref allele is 1
KnownHaps = invertArray(KnownHaps)
# Find unique haplotypes
inHapArray, UniqueNames = UniqueHaps(KnownHaps, KnownNames)
# Count number of unique haplotypes
numHapsInitial = len(UniqueNames)
# Count number of SNPs
numSNPs = inHapArray.shape[0]
```

```
# Add "dummy" SNP to ensure haplotype frequencies sum correctly

# Write out starting nexus files for comparison to endpoints
NexusWriter(KnownNames, KnownHaps, numSNPs, o.outPrefix,
            "INITIAL", o.knownHaps)
NexusWriter(UniqueNames, inHapArray, numSNPs, o.outPrefix,
            "Unique1", o.knownHaps)

# Load SNPs
finSNPs, poolNames = toNP_array(o.inFreqs, "RF")
# Add "dummy" SNP to ensure haplotype frequencies sum correctly
finSNPs = ExtendHaps(finSNPs)
# Count number of pools present
numPools = len(poolNames)
# Convert haplotypes and snps arrays to decimal format to prevent rounding
# errors
finSNPs = npToDecNp(finSNPs)

# Output for random orders (this will get updated as I figure out sorting
# and haplotype selection)
# Output haplotypes for each random order, along with RSS values for those
# haplotypes
rawOutput = open("%s_RAW.csv" % outPrefix, 'wb')
rawOutput.write("Ordering,Solution,RSS, Haplotypes")
for randIter in xrange(len(cleaned)):
    for solIter in xrange(len(cleaned[randIter])):
        rawOutput.write(
            "\n%s,%s,%s,%s" % (
                str(randIter),
                str(solIter),
                str(cleaned[randIter][solIter][1]),
                ",".join([str(x) for x in cleaned[randIter][solIter][0]])
                )
            )
rawOutput.close()

# Output frequencies for each haplotype across all orders
# This part will probably stay and be used in sorting haplotypes eventually
print("Creating summary outputs")
summaryOutput = open("%s_summary.csv" % outPrefix, 'wb')
summaryOutput.write("Haplotype,Frequency")
haplotypeCounter = {}
for randIter in xrange(len(cleaned)):
    tmpCounter = {}
    for solIter in xrange(len(cleaned[randIter])):
        for hapIter in  cleaned[randIter][solIter][0]:
            if hapIter in tmpCounter.keys():
                tmpCounter[hapIter] += 1.
            else:
                tmpCounter[hapIter] = 1.
    for keyIter in tmpCounter.keys():
        if keyIter in haplotypeCounter.keys():
            haplotypeCounter[keyIter] += tmpCounter[
                keyIter]/len(cleaned[randIter])
        else:
            haplotypeCounter[keyIter] = tmpCounter[
                keyIter]/len(cleaned[randIter])
for keyIter in haplotypeCounter.keys():
    summaryOutput.write(
        "\n%s,%s" % (
```

```
            str(keyIter), str(haplotypeCounter[keyIter] /len(cleaned))
            )
        )
summaryOutput.close()

## Group solutions into unique solutions
print("Find unique topologies")
UniqueTopologies = []
# Use sets for sorting to keep different oreders of the same haplotypes
# from being called different topologies
UniqueTopoSets = []
UniqueTopoRSSs = []
countTopoOccurances = []
for randIter in xrange(len(cleaned)):
    numSols = len(cleaned[randIter])
    for solIter in xrange(numSols):
        if set(cleaned[randIter][solIter][0]) not in UniqueTopoSets:
            UniqueTopologies.append(cleaned[randIter][solIter][0])
            UniqueTopoSets.append(set(cleaned[randIter][solIter][0]))
            UniqueTopoRSSs.append(cleaned[randIter][solIter][1])
            countTopoOccurances.append(1./numSols)
        else:
            countTopoOccurances[UniqueTopoSets.index(
                set(cleaned[randIter][solIter][0])
                )] += 1./numSols
topoCountsOutput = open("%s_topologies.csv" % outPrefix, 'wb')
topoCountsOutput.write("RSS,Occurances,Haplotypes")

# Output counts for different topologies
for topoIter in xrange(len(UniqueTopologies)):
    topoCountsOutput.write(
        "\n%s,%s,%s" % (
            UniqueTopoRSSs[topoIter],
            countTopoOccurances[topoIter],
            ",".join([str(x) for x in UniqueTopologies[topoIter]])
            )
        )
topoCountsOutput.close()

## Sort unique solutions by RSS value
# Sort a list of pointers by RSS values they point to
# This is a list of indexes to UniqueTopologies and UniqueTopoRSSs
print("Sort by RSS")
RssPointers = sorted(range(len(UniqueTopoRSSs)),
                     key=lambda x: UniqueTopoRSSs[x])
## Find the third best RSS value
# Keep track of if which RSS value this is
whichBest = 1
bestRSS = UniqueTopoRSSs[RssPointers[0]]
currPointer = 1
while currPointer < len(RssPointers) and whichBest <= o.topNum:
    if UniqueTopoRSSs[RssPointers[currPointer]] > bestRSS:
        whichBest += 1
        bestRSS = UniqueTopoRSSs[RssPointers[currPointer]]
    currPointer += 1
## Pull out the haplotype sets with one of the top three RSS values
## For each haplotype set:
finTopos = []
finDecHaps = []
print("Extract solutions from best RSS values")
```

```
for convPointer in xrange(currPointer):
    # Convert haplotype set to list of numpy arrays
    finTopos.append(
        [DecHapToNPHap(UniqueTopologies[RssPointers[convPointer]][x])
        for x in xrange(len(UniqueTopologies[RssPointers[convPointer]]))]
        )
    finDecHaps.append(
        [UniqueTopologies[RssPointers[convPointer]][x]
        for x in xrange(len(UniqueTopologies[RssPointers[convPointer]]))]
        )


## For each converted haplotype set:
    ## Find best solution for this haplotype set
    ## Output this solution as all requested outputs
print("Find haplotype frequencies and output files")
for outTopoPtr in xrange(len(finTopos)):
    # Create final haplotypes array
    finSolution = np.concatenate(
        [finTopos[outTopoPtr][x][np.newaxis].transpose()
        for x in xrange(len(finTopos[outTopoPtr]))], axis=1
        )
    # Find (or make) haplotype names
    myHapNames = []
    print("Outputing solution %s/%s" % (str(outTopoPtr + 1),
                                        str(len(finTopos))))
    print("Finding haplotype names...")
    for haplotypeIter in xrange(finSolution.shape[1]):
        if haplotypeIter >= len(UniqueNames):
            # For new haplotypes, build a new haplotype name, keeping track
            # of iteration and new haplotype number
            myHapNames.append("NewHap_%s" % (
                str(finDecHaps[outTopoPtr][haplotypeIter]))
                )
        else:
            # For known haplotypes, use the original haplotype name
            myHapNames.append(UniqueNames[haplotypeIter])
    # If requested, generate a structure formatted file
    if o.strOutput:
        outFile = open("%s_%s.str" % (outPrefix, outTopoPtr), 'wb')
    # Generate the haplotype frequencies file
    outFile2 = open("%s_%s_freqs.csv" % (outPrefix, outTopoPtr), 'wb')
    outFile2.write("Population,")
    # Create decimal haplotype identifiers
    # Finish writing first line of haplotype frequencies file
    outFile2.write(",".join(myHapNames))
    outFile2.write(",RSS")
    # Write decimal names of haplotypes
    outFile2.write(
        "\n,%s" % ",".join([str(x) for x in finDecHaps[outTopoPtr]])
        )
    # Create genpop output, if requested
    if o.genpopOutput:
        genpopOut = open("%s_%s.genpop" % (outPrefix, outTopoPtr), 'wb')
        genpopOut.write(
            ",%s" % (",".join(["cp." + str(x)
                               for x in finDecHaps[outTopoPtr]]))
            )
    SLSqs = []
    Freqs = []
```

```
predSnpFreqs = []
# Create regression output
regressionOutput = open(
    "%s_%s_Regression.csv" % (outPrefix, outTopoPtr), 'wb'
    )
regressionOutput.write(
    "Pool,SNP,Observed Frequency,Predicted Frequency\n"
    )
# Create predicted frequencies VCF output
output3 = vcfWriter(
    "%s_%s_PredFreqs.vcf" % (outPrefix, outTopoPtr),
    source="CallHaps_HapCallr_%s" % progVersion)
output3.writeHeader(poolNames)
output3.setFormat("RF")

tmpVCF = vcfReader(o.knownHaps)
output3.importLinesInfo(
    tmpVCF.getData("chrom", lineTarget="a"),
    tmpVCF.getData("pos", lineTarget="a"),
    tmpVCF.getData("ref", lineTarget="a"),
    tmpVCF.getData("alt", lineTarget="a"),
    tmpVCF.getData("qual", lineTarget="a")
    )
newResiduals = []
print("Finding haplotype frequencies...")
for poolIter in xrange(numPools):
    tmpSol = Find_Freqs(finSolution, finSNPs[:,poolIter], o.poolSize)
    SLSqs.append(tmpSol[1])
    Freqs.append(tmpSol[0])
    # Write haplotype frequencies and RSS values for this pool
    outFile2.write(
        "\n%s,%s" % (poolNames[poolIter],
                     ",".join([str(x) for x in tmpSol[0][0]]))
        )
    outFile2.write(",%s" % tmpSol[1])
    # Write genpop file text for this pool, if requested
    if o.genpopOutput:
        genpopOut.write(
            "\n%s,%s" % (poolNames[poolIter],
                         ",".join([str(x) for x in tmpSol[0][0]]))
            )
    # Write structure file text for this pool, if requested
    if o.strOutput:
        outputProt(UniqueNames, tmpSol[0], finSolution, o.poolSize,
                   poolNames, poolIter, outFile)
    # Calculate residuals for this pool
    newResiduals.append(
        np.array([[x] for x in list(residuals(tmpSol[0][0],
                                              finSolution,
                                              finSNPs[:,poolIter],
                                              o.poolSize))])
        )
    # Calculate predicted SNP frequencies
    predSnpFreqs = np.sum(
        finSolution * tmpSol[0][0], axis = 1
        )/o.poolSize
    #print("##DEBUG")
    # Write regression file lines for this pool
    regOutLines = zip(
        [poolNames[poolIter]
```

```
            for x in xrange(len(predSnpFreqs))],
        [str(y) for y in xrange(len(predSnpFreqs))],
        [str(z) for z in list(finSNPs[:,poolIter])],
        [str(w) for w in list(predSnpFreqs)]
        )
    regressionOutput.write(
        "\n".join([",".join(regOutLines[x])
                  for x in xrange(len(regOutLines))])
        )
    regressionOutput.write("\n") # add a new line between pools
    # Add predicted SNP frequencies to VCF output
    output3.importSampleValues(list(predSnpFreqs), poolNames[poolIter])
# Calculate per SNP RSS values for VCF output
SnpResiduals = [float(sum([newResiduals[poolIter][x]**2
                           for poolIter in xrange(numPools)])[0])
                for x in xrange(numSNPs)]
output3.importInfo("RSS",SnpResiduals)
output3.writeSamples()
# Close output files
output3.close()
regressionOutput.close()
outFile2.close()
if o.strOutput:
    outFile.close()
if o.genpopOutput:
    genpopOut.close()
# Write Nexus file for this solution
# This allows for network phylogeny construction
NexusWriter(myHapNames, finSolution, numSNPs, outPrefix,
            outTopoPtr, o.knownHaps)
```

Modules/CallHap_LeastSquares.py

```
#!/usr/bin/env python
# CallHap CallHap_LeastSquares.py
# By Brendan Kohrn
# 3/20/2017
#
# The main Sum Least Squares method for CallHap_HapCallr

import numpy as np
import decimal as dec
from General import *

def Find_Freqs(A, b, p):
    '''Find the frequency of various haplotypes in a pool.  A is the haplotypes
    matrix, b is the SNP Frequency matrix, and p is pool size'''
    # Set variables for number of haplotypes and number of SNPs
    M = A.shape[0]
    N = A.shape[1]
    # Create an empty numpy array for the current solution
    x = npDecZeros(1, N)
    # Create an empty numpy array to hold the last solution
    lastX = npDecZeros(1, N)
    # Create dummy variables to hold the last and current sum squared residuals
    currentSSR = -1
    lastSSR = -1

    # Run the first test to determine the best starting haplotype
    currentSSR, x = InitialTest(A, b, x, currentSSR, M, N, p)
    lastSSR = currentSSR
    lastX = np.copy(x)
    # Create finished switch and counter to check for infinite loops
    finished = False
    # Iterations:
    while not finished:
        # invoke the mail loop
        currentSSR, x = mainLoop(A, b, x, N, M, currentSSR, p)
        # If the SSR (Sum Squared Residuals; equivalent to RSS) value increases
        # on this loop, finish
        if currentSSR >= lastSSR:
            finished = True
        else:
            lastSSR = currentSSR
            lastX = np.copy(x)

    # output frequencies are contained in lastX
    # output SSR contained in lastSSR
    return(lastX, lastSSR)

def InitialTest(A, b, xIT, curSSR, M, N, p):
    # set counter for best sum squared residuals
    bestSSR = -1
    # Check each haplotype
    for hapIndex in xrange(N):
        # Calculate the SSR if this haplotype was the only one in the pool
        testSSR = sum([resid**2 for resid in np.subtract(A[:, hapIndex], b)])
        # Check if this SSR is an improvement
        if bestSSR == -1:
            bestSSR = [hapIndex, testSSR]
        elif testSSR < bestSSR[1]:
```

```
            bestSSR[0] = hapIndex
            bestSSR[1] = testSSR
    xIT[0, bestSSR[0]] = dec.Decimal(p)
    # Return SSR value and best haplotype frequency vector
    return(bestSSR[1], np.copy(xIT))

def SSR(A, xSSR, b):
    '''Calculate the sume of squared residuals for a given solution to Ax=b'''
    if type(b) == list:
        out = sum([resid**2 for resid in np.subtract(np.sum(A * xSSR, 1), b)])
    else:
        out = sum([resid**2 for resid in np.subtract(np.sum(A * xSSR, 1),
                                                     b.ravel())])
    return(out)

def mainLoop(A, b, xML, N, M, currSSR, p):
    # for each element of x s.t. x[x_1] > 1, subtract 1 from that element
    # and add one to each other element (x_2) in turn;
    bestSSR = [(-1,-1),currSSR]
    for x_1 in xrange(N):
        if xML[0, x_1] > 1:
            for x_2 in xrange(N):
                wx = np.copy(xML)
                if x_1 != x_2:
                    wx[0, x_1] -= 1
                    wx[0, x_2] += 1
                    testSSR = SSR(A, wx / p, b)
                    if testSSR < bestSSR[1]:
                        bestSSR[0] = (x_1, x_2)
                        bestSSR[1] = testSSR
    if bestSSR[0] != (-1,-1):
        xML[0,bestSSR[0][0]] -= 1
        xML[0,bestSSR[0][1]] += 1
    return(bestSSR[1], np.copy(xML))
```

Modules/General.py

```
#!/usr/bin/env python
# CallHap General.py
# By Brendan Kohrn
# 3/20/2017
#
# This script contains general functions for CallHap

import numpy as np
import math
import decimal as dec

def comparePotHaps(potHapSetA, potHapSetB, numInitialHaps):
    '''Check if all haplotypes in two haplotype sets are the same'''
    # If two haplotype sets are different lengths, they are different
    if len(potHapSetA) != len(potHapSetB):
        return(False)
    else:
        return(all(np.all(x==y) for x,y in zip(potHapSetA[numInitialHaps:],
                                               potHapSetB[numInitialHaps:])))

def average(inList):
    ''' Take the average value of a list'''
    # Make sure the list has length
    if len(inList) == 0:
        raise("Error in Average: %s" % inList)
    return(float(sum(inList))/len(inList))

def npDecZeros(rows, cols=0):
    '''Create a numpy array of Decimal(0) values'''
    if cols == 0:
        outArray = np.zeros(rows,dtype=dec.Decimal)
        for rowIter in xrange(rows):
            outArray[rowIter] = dec.Decimal(outArray[rowIter])
    else:
        outArray = np.zeros((rows,cols), dtype=dec.Decimal)
        for rowIter in xrange(rows):
            for colIter in xrange(cols):
                outArray[rowIter,colIter] = dec.Decimal(outArray[rowIter,
                                                                 colIter])

    return(outArray)

def npToDecNp(inArray):
    '''Convert a numpy array of floats to a numpy array of Decimal numbers to
    avoid rounding errors'''
    outArray = np.array(inArray, dtype=dec.Decimal)
    for elmnt, value in np.ndenumerate(outArray):
        outArray[elmnt] = dec.Decimal(outArray[elmnt])
    return(outArray)

def copy(inArr, elmntType = "int"):
    '''Copy a list (particularly of numpy arrays).'''
    if elmntType == "nparray":
        return([np.copy(x) for x in inArr])
    else:
        return([x for x in inArr])

def AIC_from_RSS(RSS, numHaps, numSNPs):
```

```
    '''Calculate AIC from RSS values'''
    AIC = 2 * numHaps + (numSNPs * math.log10(RSS/numSNPs))
    return(AIC)

def AICc_from_RSS(RSS, numHaps, numSNPs):
    '''Calculate AICc from RSS values'''
    AIC = 2 * numHaps + (numSNPs * math.log10(RSS/numSNPs)) + (2*numHaps *
        (numHaps + 1))/(numSNPs - numHaps - 1)
    return(AIC)

def invertArray(inArray):
    '''Invert an array of 0s and 1s (such as the Haplotypes array) or an array
    between 0 and 1 (such as the SNP Freqs array).'''
    OutArray = 1 - inArray
    return(OutArray)

def residuals(inSol, inData, inFreqs, poolSize):
    '''Calculate residuals for one particular least-squares solution of Ax=b'''
    calculated = np.sum((inSol * inData)/poolSize, 1)
    resid = np.subtract(inFreqs, calculated)
    return(resid)

def ArrayHaps(origHaps, newHaps):
    allHapsToArray = [origHaps]
    allHapsToArray.extend(newHaps)
    return(np.concatenate(allHapsToArray, axis=1))

def numDiffs(inHap1, inHap2):
    if inHap1.shape != inHap2.shape:
        raise
    else:
        in1 = inHap1.ravel()
        in2 = inHap2.ravel()
        diffCounter = sum([0 if in1[x] == in2[x] else 1
                           for x in xrange(len(in1)) ])
        return(diffCounter)

def areEqual(inHap1, inHap2):
    if inHap1.shape != inHap2.shape:
        return(False)
    else:
        return(np.all(inHap1 == inHap2))

def FindLegalSnpsByNetwork(inHaps, testHapIdx):
    closestHaps = []
    closestDiffs = []
    notClosest = []
    numSnps = len(inHaps[0])
    distances=[numSnps - np.sum(a==inHaps[testHapIdx]) for a in inHaps]
    # Determine the distance between this haplotype and every other haplotype
    # in number of SNPs different
    # Sort by closeness
    distIters = sorted(range(len(distances)), key=lambda x: distances[x])
    # For each haplotype, from closest to furthest away, check if it shares a
    # difference in the target SNP
    # with another haplotype in closestHaps
    for hapIter in distIters:
        if hapIter != testHapIdx:
            if closestHaps == []:
                # If no haplotype is closest yet, this one is the closest
```

```
                closestHaps.append(hapIter)
                closestDiffs.append([])
                notClosest.append([])
                for x in xrange(numSnps):
                    if inHaps[hapIter][x] == inHaps[testHapIdx][x]:
                        pass
                    else:
                        closestDiffs[-1].append(x)
            else:
                # Otherwise, test to see if this haplotype shares a different
                # SNP with any closer haplotype
                diffBranch = True
                for closeHap in xrange(len(closestHaps)):
                    for difSnp in xrange(len(closestDiffs[closeHap])):
                        tmpPointer = closestDiffs[closeHap][difSnp]
                        if (inHaps[hapIter][tmpPointer] ==
                                inHaps[testHapIdx][tmpPointer]):
                            notClosest[closeHap].append(difSnp)
                        else:
                            diffBranch = False
                if diffBranch:
                    closestHaps.append(hapIter)
                    notClosest.append([])
                closestDiffs.append([])
                for x in xrange(numSnps):
                    if inHaps[hapIter][x] == inHaps[testHapIdx][x]:
                        pass
                    else:
                        closestDiffs[-1].append(x)
        CanChange = []
        for hap in xrange(len(closestHaps)):
            CanChange.extend(closestDiffs[hap])
    return(closestHaps[0],CanChange)

def ValidSnpsFromPhylogeny(inHaps):
    countDiffs = [[(a!=b) for a in inHaps] for b in inHaps]
    diffSnps = [[[b for b in xrange(len(countDiffs[x][a]))
                  if countDiffs[x][a][b] == True]
                  for a in xrange(len(countDiffs[x]))]
                  for x in xrange(len(countDiffs))]
    # Find adjacent haplotypes for each haplotype
    validSnps = []
    nextHaps = []
    for hap in xrange(len(inHaps)):
        nextHaps.append([])
        validSnps.append([])
        minDistOrder = sorted(range(len(inHaps)),
                              key=lambda x: len(diffSnps[hap][x]))
#        print("DEBUG")
        for hap2 in minDistOrder:
            if hap != hap2:

                if len(nextHaps[-1]) > 0:
                    isAdj = True
                    for closeHap in nextHaps[-1]:
                        if len(np.intersect1d(diffSnps[hap][closeHap],
                                              diffSnps[hap][hap2])) != 0:
                            isAdj = False
                            validSnps[-1] = list(np.setdiff1d(validSnps[-1],
                                diffSnps[closeHap][hap2]))
```

```
                    if isAdj == True:
                        nextHaps[-1].append(hap2)
                        validSnps[-1].extend(diffSnps[hap][hap2])
                else:
                    nextHaps[-1].append(hap2)
                    validSnps[-1].extend(diffSnps[hap][hap2])
    return(validSnps)

def DecHapToNPHap(decHap):
    '''Convert a decimal haplotype back into a numpy array'''
    binHap = bin(decHap)[2:]
    binHap = np.array([dec.Decimal(x) for x in binHap[1:]])
    return(binHap)
```

Modules/IO.py

```
#!/usr/bin/env python
# CallHap IO.py
# By Brendan Kohrn
# 3/20/2017
#
# This script contains functions relating to input processing of matrices
# As well as functions relating to some specific output formats.

import numpy as np
from VCF_parser import *

def ExtendHaps(origHaps):
    '''Function to add "Dummy" SNP to array; designed to ensure that all
    haplotype frequencies sum to 20'''
    allHapsToArray = [origHaps]
    allHapsToArray.extend([np.array([[1
        for x in xrange(int(origHaps.shape[1]))]])])
    return(np.concatenate(allHapsToArray, axis=0))

def UniqueHaps(inHaps, inNames):
    '''Find unique haplotypes and reduce the haplotypes and their names;
    still needs some work to fix merged names'''
    remove = [False for n in range(int(inHaps.shape[1]))]
    for iterx in range(int(inHaps.shape[1])-1):
        for y in range(iterx+1, inHaps.shape[1]):
            if not remove[y]:
                if np.ma.all(inHaps[:,iterx] == inHaps[:,y]):
                    remove[y] = True
    toRemove = [iterx for iterx in range(len(remove)) if remove[iterx]]
    names = [inNames[iterx] for iterx in range(len(inNames))
        if not remove[iterx]]
    return(np.delete(inHaps, toRemove, 1), names)

def outputProt(UniqueNames, bestFreqs, bestArray, poolSize,
               poolNames, population, outFile):
    '''Output STRUCTURE format file text for a given population.  Call multiple
    times to create complete STRUCTURE file.'''
    decHaps = []
    hapNames = UniqueNames[:]
    newHapNumber = 1
    # Create haplotype names
    for haplotypeIter in xrange(len(bestFreqs[0])):
        if haplotypeIter >= len(UniqueNames):
            hapNames.append("NewHap_%s" % str(newHapNumber).zfill(2))
            newHapNumber += 1
        # Create decimal haplotypes
        decHaps.append(int("1"+"".join([str(int(x))
            for x in bestArray[:, haplotypeIter]]),2))
    indivHaps = []
    # Create individuals
    for haplotypeIter in xrange(len(bestFreqs[0])):
        for indivIter in xrange(int(bestFreqs[0][haplotypeIter])):

            indivHaps.append(decHaps[haplotypeIter])
    # Output individuals
    for individual in xrange(poolSize):
        outLine = " ".join(["%s_%s" % (poolNames[population],
            str(individual).zfill(len(str(poolSize)))), str(population),
```

```
            str(indivHaps[individual])])
        outFile.write("%s\n" % outLine)

def NexusWriter(myHapNames, finSolution, numSNPs, outPrefix, outIdx,
                knownHaps, snpsToRemove=[]):
    '''Output a NEXUS file for a given solution'''
    # Open a NEXUS output file
    outFile3 = open("%s_%s_haps.nex" % (outPrefix, outIdx), 'wb')
    # Write header lines
    outFile3.write("##NEXUS\n")
    outFile3.write("Begin Data;\n")
    outFile3.write("\tDimensions ntax=%s nchar=%s;\n" %
        (finSolution.shape[1], numSNPs))
    outFile3.write("\tFormat datatype=DNA missing=N gap=-;\n")
    outFile3.write("\tMatrix\n")
    # Open the initial VCF to get the ref and alt states for each SNP
    finVCF = vcfReader(knownHaps)
    refAlleles = []
    altAlleles = []
    for line in finVCF.lines:
        if line.getData("pos") not in snpsToRemove:
            refAlleles.append(line.getData("ref"))
            altAlleles.append(line.getData("alt")[0])
    # Create the output haplotype sequence by concentrating the relevant
    #alleles for each SNP, for each haplotype
    for hap in xrange(len(myHapNames)):
        outFile3.write("%s\t%s\n" % (myHapNames[hap], "".join([refAlleles[x]
            if finSolution[x, hap] == 1 else altAlleles[x]
            for x in xrange(numSNPs)])))
    outFile3.write(";\n")
    outFile3.write("End;\n")
    outFile3.close()
```

Modules/VCF_parser.py

```
#!/usr/bin/env python
# CallHap IO.py
# By Brendan Kohrn
# 3/21/2017
#
# This is the VCF parser used by all CallHap specific programs
import numpy as np
import time

class vcfFile:
    def __init__(self, inFileName, mode, source=""):
        if mode == 'r':
            return(vcfReader(inFileName))
        elif mode == 'w':
            return(vcfWriter(inFileName,source))

class vcfWriter:
    '''Class to write VCF output based on an imput template.
    Call order:
    a = vcfWriter(inName, source)
    a.writeHeader(sampNames)
    a.setFormat(formatStr)
    a.importLinesInfo(Chroms, Poss, Refs, Alts, Quals)
    for sampleName in sampNames:
        a.importSampleValues(inValues, sampleName)
    a.writeSamples()
    a.close()
    '''
    def __init__(self, inFileName, source):
        '''Initialize the class'''
        # Open an output file
        self.outputFile = open(inFileName, "wb")
        # Write header lines
        self.outputFile.write("##fileformat=VCFv4.2\n")
        self.outputFile.write("##fileDate=%s\n" % time.strftime("%Y%m%d"))
        self.outputFile.write("##source=%s\n" % source)

    def writeHeader(self, sampleNames):

        # Write column nanes line
        self.outputFile.write("#CHROM\tPOS\tID\tREF\tALT\tQUAL\t")
        self.outputFile.write("FILTER\tINFO\tFORMAT\t")
        self.outputFile.write("%s\n" % "\t".join(sampleNames))
        # Create output columns
        self.outputCols = {x:[] for x in sampleNames}
        # Save sample names
        self.sampleNames = sampleNames
        # Create counter to keep track of how many columns have been filled
        self.colsFilled = 0
        # How many columns need to be filled
        self.totalCols = len(sampleNames)

    def setFormat(self, formatStr):
        # Set the format string for outputs.
        self.formatStr = formatStr

    def importInfo(self, InfoField, InfoValues):
        '''Add text to the info field'''
```

```
        # Check that there is an info value for each row
        if len(InfoValues) != self.numRows:
            raise Exception
        # Check if there is already info data present
        if self.infosSet == True:
            # If info data is present, add to it
            for x in xrange(self.numRows):
                self.infos[x] += ";%s=%s" % (InfoField, InfoValues)
        else:
            # Create info data
            self.infos = ["%s=%s" % (InfoField, InfoValues[x])
                          for x in xrange(self.numRows)]
            self.infosSet = True

    def importLinesInfo(self, Chroms, Poss, Refs, Alts, Quals,
                        IDs = None, Filts = None, Infos = None):
        '''Add positional and quality information about the lines'''
        testLen = len(Chroms)
        # Check that all lists of values are the same length
        if len(Poss) != testLen:
            raise Exception
        elif len(Refs) != testLen:
            raise Exception
        elif len(Alts) != testLen:
            raise Exception
        elif len(Quals) != testLen:
            raise Exception
        elif IDs != None:
            if len(IDs) != testLen:
                raise Exception
        elif Filts != None:
            if len(Filts) != testLen:
                raise Exception
        elif Infos != None:
            if len(Infos) != testLen:
                raise Exception
        self.numRows = testLen
        self.chroms = Chroms
        self.pos = Poss
        if IDs == None:
            self.ID_Set = False
            self.IDs = ['.' for x in xrange(len(Chroms))]
        else:
            self.ID_Set = True
            self.IDs = IDs
        self.refs = Refs
        self.alts = [x[0] for x in Alts]
        self.quals = Quals
        if Filts == None:
            self.filts = ['.' for x in xrange(len(Chroms))]
        else:
            self.filts = Filts
        if Infos == None:
            self.infosSet = False
            self.infos = ['.' for x in xrange(len(Chroms))]
        else:
            self.infos = Infos

    def importSampleValues(self, inValues, sampleName):
        '''Import cell data for one column of a VCF file'''
```

```
            # Old debugging text
            if len(inValues) != self.numRows + 1:
                print(type(inValues))
                print(len(inValues))
                print(inValues)
                print(self.numRows)
                raise Exception
            # Fill the column
            self.outputCols[sampleName] = inValues[:-1]
            self.colsFilled += 1

        def removeRows(self, rowsToRemove):
            '''Remove rows from the VCF output'''
            removalRows = sorted(rowsToRemove, reverse = True)
            for rowIter in removalRows:
                self.chroms.pop(rowIter)
                self.pos.pop(rowIter)
                self.IDs.pop(rowIter)
                self.refs.pop(rowIter)
                self.alts.pop(rowIter)
                self.quals.pop(rowIter)
                self.filts.pop(rowIter)
                self.infos.pop(rowIter)
                self.numRows -= 1
            self.skippedRows = rowsToRemove

        def writeSamples(self):
            '''Write the VCF output to file'''
            # Throw an error if not all columns have been filled
            if self.colsFilled != self.totalCols:
                raise Exception
            else:
                for lineNum in xrange(self.numRows):
                    outLine = "%s\t" % self.chroms[lineNum]
                    outLine += "%s\t" % self.pos[lineNum]
                    outLine += "%s\t" % self.IDs[lineNum]
                    outLine += "%s\t" % self.refs[lineNum]
                    outLine += "%s\t" % self.alts[lineNum]
                    outLine += "%s\t" % self.quals[lineNum]
                    outLine += "%s\t" % self.filts[lineNum]
                    outLine += "%s\t" % self.infos[lineNum]
                    outLine += "%s\t" % self.formatStr
                    outLine += "%s\n" % "\t".join(
                        [str(self.outputCols[self.sampleNames[x]][lineNum])
                         for x in xrange(len(self.sampleNames))]
                         )
                    self.outputFile.write(outLine)

        def close(self):
            '''Close the output file'''
            self.outputFile.close()

    class vcfReader:
        '''Method for reading VCF files'''
        def __init__(self, inFileName):
            '''Initialize the reader and read the file'''
            self.headInfo = {}
            self.lines = []
            # Open the file
            inFile = open(inFileName, "rb")
```

```
            for line in inFile:
                # Check if this is a header line
                if line[0:2] == "##":
                    # Parse the header line, in case that information is needed
                    # later
                    wLine = line.strip("#").split("=", 1)
                    if "INFO" in wLine[1]:
                        if "INFO" not in self.headInfo:
                            self.headInfo["INFO"] = {}
                        linebins = wLine[1].strip("<>").split(",")
                        self.headInfo["INFO"][linebins[0].split("=")[1]] = {
                            x.split("=")[0]: x.split("=")[1] for x in linebins
                            }

                    elif "FORMAT" in wLine[1]:
                        if "FORMAT" not in self.headInfo:
                            self.headInfo["FORMAT"] = {}
                        linebins = wLine[1].strip("<>").split(",")
                        self.headInfo["FORMAT"][linebins[0].split("=")[1]] = {
                            x.split("=")[0]: x.split("=")[1] for x in linebins
                            }
                    else:
                        self.headInfo[wLine[0]] = wLine[1]
                # Check if this is the column labels line
                elif line[0] == "#":
                    # Save the sample names
                    self.sampNames = line.strip().split()[9:]
                else:
                    # Create a new VCF line with the data in this line
                    self.lines.append(vcfLine(line))
            # Close the input file
            inFile.close()

        def getData(self, target, lineTarget = None, sampTarget = None):
            '''Retrieve data about the VCF file from a specific line or information
             column'''
            if target in ("chrom", "pos", "ID", "ref", "alt",
                          "qual", "filt","info", "form"):
                if lineTarget == 'a':
                    outList = []
                    for line in self.lines:
                        outList.append(line.getData(target))
                    return(outList)

        def getNames(self):
            '''Retrieve the column names'''
            return(self.sampNames)

        def asNumpyArray(self, target):
            '''Output the data from this file as a numpy array'''
            # This method assumes targeting all rows and columns
            prearray = [x.toNP(target) for x in self.lines]
            return(np.array(prearray))

    class vcfLine:
        '''Handler class for a single line of a VCF file'''
        def __init__(self, inLine):
            '''Initialize the class and read in the data'''
            linebins = inLine.split()
            self.data = {}
```

```
        self.data["chrom"] = linebins[0]
        self.data["pos"] = int(linebins[1])
        self.data["ID"] = linebins[2]
        self.data["ref"] = linebins[3]
        self.data["alt"] = linebins[4].split(",")
        self.data["qual"] = linebins[5]
        self.data["filt"] = linebins[6]
        if linebins[7] == ".":
            self.data["info"] = {}
        else:
            self.data["info"] = {a.split("=")[0]: a.split("=")[1]
                                 for a in linebins[7].split(";")}
        self.data["form"] = linebins[8].split(":")
        # Save the data from each column within this row as a VCF cell class
        self.data["data"] = [vcfCell(self.data["form"], a)
                             for a in linebins[9:]]

    def getData(self, target, sampTarget = None):
        '''Retrieve data from this line'''
        if sampTarget == None and target in self.data.keys():
            return self.data[target]
        elif sampTarget == "a":
            return([self.data["data"][x].getData(target)
                    for x in xrange(len(self.data["data"]))])
        elif "info" in sampTarget:
            if ":" in sampTarget:
                infoTarget = sampTarget.split(":")[1]
                if infoTarget in self.data["info"].keys():
                    return(self.data["info"][infoTarget])
                else:
                    raise
            else:
                return(self.data["info"])
        elif sampTarget < len(self.data["data"]):
            return(self.data["data"][sampTarget].getData(target))
        else:
            raise

    def toNP(self, target):
        '''Get the data from this row to for numpy array creation'''
        return([self.data["data"][x].toNP(target) for x in
xrange(len(self.data["data"]))])

    def setElmt(self, target, newValue):
        '''Set a specific value in the data from this row'''
        if target in self.data.keys():
            self.data[target] = newValue
        else:
            raise

class vcfCell:
    '''Class for holding data from a single cell of a VCF file'''
    def __init__(self, FormatList, inCellText):
        '''Create the cell'''
        cellbins = inCellText.split(":")
        if cellbins[0] == ".":
            self.data = {x: [np.nan] for x in FormatList}
        else:
            self.data = {}
            for formatIter in xrange(len(FormatList)):
```

```
                self.data[FormatList[formatIter]] = [float(x)
                    for x in cellbins[formatIter].split(",")]

    def getData(self, target=None):
        '''Retrieve data from the cell'''
        if target == None:
            return(self.data)
        elif target in self.data.keys():
            return(self.data[target][0])
        else:
            raise

    def toNP(self, target):
        '''Get data from the cell to create a numpy array'''
        return(float(self.data[target][0]))

def toNP_array(inFileName, target):
    '''Open a VCF file and create a numpy array of a specific type of data from
     that array, along with the sample names'''
    tmpVCF = vcfReader(inFileName)
    output = tmpVCF.asNumpyArray(target)
    out2 = tmpVCF.getNames()
    return(output, out2)
```

Modules/parallel.py

```
#!/usr/bin/env python
# CallHap parallel.py
# By Brendan Kohrn
# 3/20/2017
#
# This script includes multiprocessing functionality for CallHap

import numpy as np
from functools import partial
from MakeHaplotypes import *
from CallHap_LeastSquares import *
from General import *
import sys
from multiprocessing import Pool

'''This module contains parallelization methods.  Some of these will no longer
   be needed in random order processing'''

def easyConcat(listHaps):
    '''One argument command for concatenating a list of arrays into a single
       array'''
    return(np.concatenate([x[np.newaxis].transpose()
           for x in listHaps], axis=1))

def massFindFreqs(inHaps, inSnpFreqs, p):
    '''Find frequencies for many pools at the same time'''
    mySLSqs = []
    myFreqs = []
    for poolIter in xrange(inSnpFreqs.shape[1]):
        tmpSol = Find_Freqs(inHaps, inSnpFreqs[:, poolIter], p)
        mySLSqs.append(tmpSol[1])
        myFreqs.append(tmpSol[0])
    myAIC = sum(mySLSqs)/len(mySLSqs)
    return(mySLSqs, myFreqs, myAIC)

def easy_parallizeLS(sequence, numProcesses, snpsFreqs, poolSize):
    '''parallelization method for finding the frequencies for several potential
    haplotype sets at the same time.
    Not used in random ordering'''
    pool = Pool(processes=numProcesses, maxtasksperchild=500)
    intermediate = pool.map(easyConcat, sequence)
    # Concatenate the haplotype sets into a single numpy array
    cleanedIntermediate = [x for x in intermediate if not x is None]
    pool.close()
    pool.join()
    pool2 = Pool(processes=numProcesses, maxtasksperchild=500)
    #Create function for single argument calling of FindFreqs
    func = partial(massFindFreqs, inSnpFreqs=snpsFreqs, p=poolSize)
    # Find haplotype frequencies for each potential haplotype set
    result = pool2.map(func, cleanedIntermediate)
    cleaned = [x for x in result if not x is None]
    # not optimal but safe
    pool2.close()
    pool2.join()
    return(cleaned)
```